Your Company Name or Logo

Design Document for:

QuadBoom

Implementing a classic puzzler on the Xbox 360

"Or, why glitches are horrendously evil, but to be completely expected"™

All work Copyright ©2010 by Erik Ottosen

Written by Erik Ottosen

Version # 1.50

New Content starts on Page 15

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

Design History

The design document is currently running on it's first revision. Any completed document revisions will have details here.

Version 1.00

Version 1.00 represents the first run-through of the Document.

1. Basic premises completed.
2. TODO: Write characters besides the sample.
3. TODO: Write story concepts
4. TODO: Research feasibility of all features

Version 1.50

Version 1.50 represents an update on the Document to account for the completion of Design Alpha One of the actual working program.

1. Updated with basic technical details, including examples of how they work in Design Alpha One.
2. Updated feasibility analysis
3. TODO: Write more characters
4. TODO: Detail story concept

All pages until page 14 are updates from the 1.00 Design Document turned in in February; Page 14 begins the elaboration of technology details.

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

Game Overview

Philosophy

Philosophical point #1

This game seeks to take the classic video game series, “Puyo Puyo”, and successfully implement it's core on the XBox 360. The game isn't seeking to be revolutionary or special, only solid. The implementation will, if completed sufficiently, go up on XBox Indie Games for sale to a potential audience of close to 20 million players.

Philosophical point #2

This game's implementation is focused on a working Xbox 360 version, but could easily be ported to Windows (at least, Windows Vista and 7) and possibly Microsoft's Zune, as well. The goal, therefore, must be to complete a game competing with other Xbox Live Indie Games, such as the hit “I MAED A GAM3 W1TH Z0MB1ES!!!1” (No, I'm not joking.)

Philosophical point #3

The game also seeks to implement some features to keep it distinct from existing games in the Puyo style – and other popular puzzlers already on the 360 – by the introduction of distinct characters, who have game-shifting special moves, a wide array of gimmick missions, and other features.

Common Questions

What is the game?

QuadBoom is a simple puzzle game of the “drop-and-match” tradition derived from Tetris. Players drop pairs (or, less frequently, triples and quads) of pieces of varying colors, making groups of four of the same color to remove the pieces. The game is generally played in a multiplayer, 'last man standing' format, where players send 'junk pieces' to one another to try and fill each other's tables.

Why create this game?

This game does not currently exist on *any* active home video game system. To complete it and thus fill the niche is not excessively difficult, but also isn't very interesting; extra features, however can help create a larger, better game, and provide an adjustable challenge.

Where does the game take place?

QuadBoom takes place in a high fantasy universe, making fun of fantasy archetypes in the tradition of series such as *Slayers.* More specifically, it takes place in Len City, a largish city famous for it's well-established adventurer's guild – and the army of people hoping to profit off of said guild's presence.

What do I control?

Players don't control a character directly *per se* – instead, they manipulate a puzzle field that represents the combat or situation they are currently encountering. Eliminating pieces from the board would thus represent attacking opponents, and pieces getting dropped into the board due to other's actions, taking damage. The fact that this is a representation is, like everything else in the game, parodied and made fun of with at least some frequency.

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

How many characters do I control?

The game's number of playable characters could vary – the goal is to have at least ten, each with unique gimmicks that modify how the game is played (most extravagently, their special moves). The exact characters are to be determined, but parodies of fantasy archetypes would be the norm.

Each character has their own plot, but not one gets to do any epic questing... Not that it'll stop at least some of them from being a Large Ham anyways.

What is the main focus?

The storylines of the game would simply consist of each character's petty concerns, running into each other and scuffles breaking out as a result. The result could either be a set of criss-crossing lines, or a series of completely separate plots. (The former might be funnier, if only so the “Final Boss” character has to keep running from place to place so he can be the Final Boss for everyone.)

What’s different?

The biggest thing to make this different from existing Puyo games, is an emphasis on character variation. All manner of normal game assumptions (such as the size of the player's field) can change heavily due to character choice, and players can get a supply of special moves that severely change play elements.

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

Feature Set

* marks a feature as not implemented in Alpha One, but intended for later builds.
** marks a feature as intended for *much* later builds, far down the line.
*** may be flat-out impossible on Xbox Indie Games.

General Features

Simple, yet addictive play
Character variations as a source of depth**
2D graphics in a style reminiscient of 16-bit games* (To wit: the graphics are not in anything resembling a 16-bit style).

Multiplayer Features

Up to four players on one Xbox 360 console*
Support for odd controllers (such as Rock Band equipment)**
Online multiplayer via Xbox LIVE, including TrueSkill ranking*
Variant modes to twist play and provide greater party fun**

Singleplayer Features

All characters available in Multiplayer have individual storylines as well**
All plots have gimmick-modified missions to twist things up**
Comedy, and lots of it (of varying quality and classiness – the goal is to have a 1/3 in all community content ratings)*

Gameplay

Simple controls – one D-Pad, two spin buttons, three special move buttons (two for selection, one for use)
Possible advanced controls mode for faster use of specials by taking advantage of the 360's two Analog Sticks**
Fast, clean basic play
Classic design that still holds up in modern play, but is easily varied upon

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

The Game World

Overview

The game takes place in Len City, the largest city (but not the capital) of the Not Evil Really Empire on the planet Japatar. (As the narration claims, “You come up with better names on short notice!”) The city is particularly famous for it's large Adventurer's Guild, and a ridiculous array of businesses gathered around it, trying to attract Adventurers... Which have created a spatial warp in their multitude. No one seems to complain.

Len City is not completely explored during the storyline; instead, Adventurer's Square, a one-square-mile block of the city, is where everything occurs. In the center of the area is the actual Adventurer's Guild, where heroes pick up epic world-saving quests... Or more modest goals. Surrounding them are a multitude of shops and 'services'. Specifics to be defined by character.

(Details on individual places of interest forthcoming in later designs; this is not critical to gameplay design)

World Layout Detail #1

World Layout Detail #2

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

Rendering System

Overview

The game uses classical 2D graphics – during menus and cutscenes, the entire scene is shown in a standard layout. In (normal) gameplay, three panes – one for each player's field, plus one in the center showing the characters and statistics like score and what pieces are coming next – provide all needed gameplay information, plus competitive graphics.

2D/3D Rendering

The default Xbox 360 rendering system proved to be more than sufficient for basic 2D gameplay with modest special effects, requiring no special code to operate.

It is expected that later builds may require the use of higher-level shaders, however, these can be easily implemented separately from the engine using the 360's Shader features.

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

Game Characters

Overview

All game characters represent varying fantasy game, anime, and novel archetypes, played entirely for laughs.

Current ideas and abilities include:

* Fighter, who is intelligent and erudite, but can't get past those stupid stereotypes.

Has a larger field than most characters; specials favor deflecting of attacks, but also includes the ability to swap two horizontally adjacent pieces.

* Magician, who just wants to enjoy a decent cake at one of the cafes. There's (#ofothercharacters) of them, but if she can just find one without a fight breaking out and destroying the cafe or getting her kicked out..

Spells that remove pieces from the field – both sides of it. While one might think that removing pieces

* The Final Boss, who practices necromancy, demonology, and jaywalking. He sticks around the Adventurer's Guild because sometimes the Guild needs to hire a Final Boss to send at an Adventurer; he makes more quickly improvising an evil lair, a few masked minions with a strict “run away at the first sign of trouble” contract, and a plastic skull throne than doing the real thing.

Has a cosmically large field and an unusually fast AI, but his specials, while very large-scale, take a long time to set up.

NPCs

A few NPCs can show up during the plot, which are not unlockable as variants for multiplayer. Most have extremely nasty benefits to their playstyle. Ironically, the game's “Final Boss” character isn't one of these.

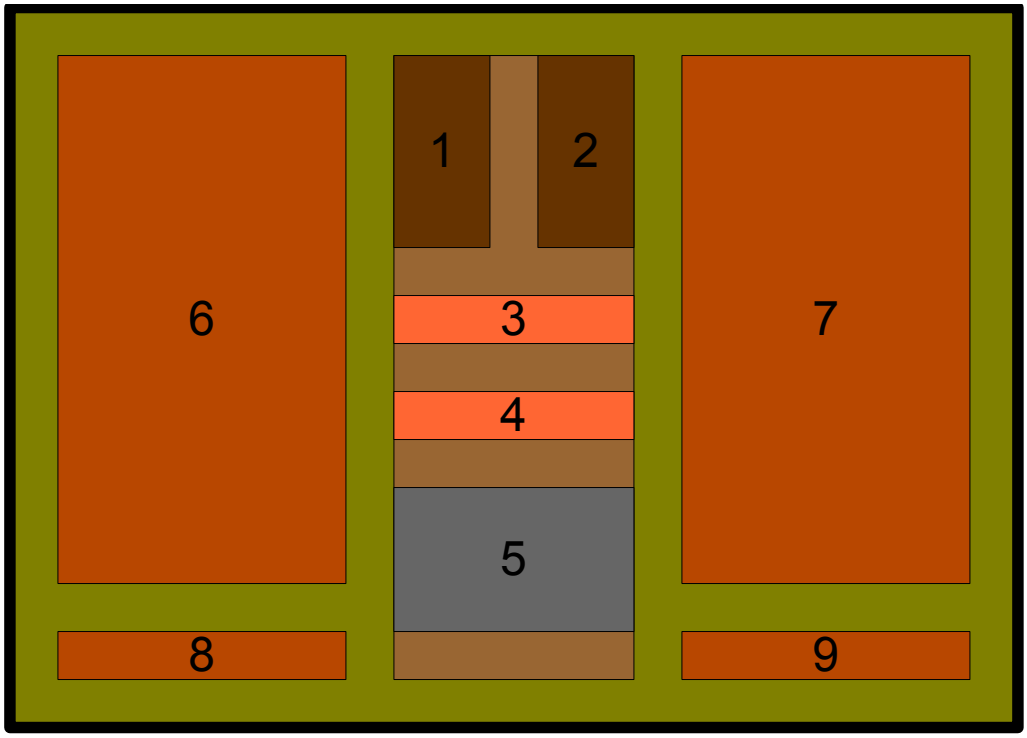
* Golem, which just stands there for three minutes before noticing the player is there... Then destroys them.

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

User Interface

Overview

Play is controlled using a standard Xbox 360 controller. (Future versions could be upgraded with a control scheme based off of Microsoft's Natal system, but it won't be available during this semester, so it can't be applied at this time.) Not counting for menus and cutscenes, all gameplay takes place on one core split-up screen. (This screen is designed to be reasonably friendly towards 4/3 screens; On the Xbox 360, you need to account for both widescreen and standard resolutions.)



1 and 2 show which piece types are falling next – the immediate next piece, and the piece after that.

3 and 4 will carry a number of circular gauges, each representing one color of piece (and going up when that piece is removed). These gauges represent the character's resources for using their special moves.

5 carries a miniature representation of each player's character, with animations for them using different techniques, being hit, etc.

6 and 7 are each player's gameplay fields – 6 units wide by 12 tall. (Players with larger fields would typically have their view 'zoomed out', making each piece smaller in order to show them all). Incoming attacks are represented in 5, and also as a small visualization on top of the pllayfields. Only field 6 is implemented in Design Alpha One, as all other fields can be considered periphereal to the main gameplay.

8 and 9 are small menus for players to select and use special moves from – using the left and right bumpers on the 360 controller to select, then pulling both the left and right triggers to activate that spell.

Empty space is primarily used for graphical elements to buffer the scene.

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

User Interface Detail #1

Characters are represented on the screen in a miniature, “SD” style during gameplay (space 5, above). Their animations usually do not directly touch the other player's character – this is meant to evoke a similar style from another classic versus puzzle game, but also makes things easier to organize technically. (A few sprite-aiming tricks for allowing attacks to visibly collide with each other to cancel out will be needed for optimization.)

User Interface Detail #2

The spellcasting interface is designed to not get in the way – all character specials need to be usable using as few buttons as possible. Some spells will require 'aiming', though – this can be done by, after casting, freezing your general play to let you aim and use the special. This allows for focused concentration, but at the cost of slowing your movements down – possibly giving an opponent a lead.

If time allows, an optional feature would allow for the spell's aiming to be done while playing, by using the 360's right analog stick, so that as soon as you cast, the target is already set. This is great for people with split attention, but comes at the cost of the opponent being able to better see what you are about to do. This would generally favor advanced players.

Space for this Interface is available for the game's final version in the present code, even after accounting for the Drop Positioning Glitch in Design Alpha One.

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

Musical Scores and Sound Effects

Overview

Design Alpha One features *no* sound code; sound code is relatively easy to integrate with XNA code, but does require special structures that would have taken significant time to learn.

The game will have very minimalistic sounds – in part because engineering good digitally-synthesized sounds is not an easy talent to find. To make up for this, talents that ARE available readily will be fully used – in particular, I can get the assistance of a few local voice talents for character sounds (though cutscenes will likely go unvoiced to favor a smaller download size), and know a fellow who should be able to produce some decent music.

Sound Design

Each character should have around twenty voice clips – four for attacking, four for being attacked, three victory and three defeat clips, and special clips for specific special actions.

Additional noises should associate with specific actions – pieces landing should have an audible 'tap', pieces being shifted from side to side or being moved down at a higher speed should have distinct, 'whoosh'-like noises, and spell casting should have a unique sound for each spell – doubling as an audio cue for the opposing player to get ready, and quickly.

Audio will be coded to allow for stereo use – playing items in the center of the screen (primarily voices) in both channels, while one channel is used for each side of the screen, so that the player sitting (presumably) on that side hears their sounds more than the other players'; this is good for providing information, and easily provided for with the XNA Sound libraries.

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

Single-Player Game

Overview

Outside of the core gameplay, the Single Player game is meant to consist of a series of campaigns; each playable character would play through anywhere from four to ten matches

Story Organization

Each player-character's story consists of five to nine cutscenes – one before each match, and one after the last match. All consist of "day in the life" segments within the city – people pursuing jobs, trying to get their payout for the jobs they finished, and the like. All the while, the resident Final Boss just wants a few hours off, but he needs the money for every job he can get, resulting in him running in a panic all over the city.

Plots can be run in any order, and don't significantly connect with one another.

Hours of Gameplay

A single campaign of this game shouldn't take much more than an hour for a player of decent skill – the point is in repeating the game regularly, both in single-player and multiplayer.

Victory Conditions

Quite simply, progress through all the matches, best the Final Boss, and watch as their character finally attains their goal! … Maybe. This being a more comedically-toned game, the results aren't always so nice...

Saving and Loading

The only data that needs to be saved or loaded in this game is (optionally) some player statistics, any high scores (if a scoring algorithm is provided; it's actually quite optional for most players, though!), and any data about unlockable items acquired (if any). Saving can be done at the time of event easily, since the resulting file is likely to run at a size of under 16 kilobytes. Loading would only need to be done if accessing certain scenes, or on program load; either way, it can be done purely 'in the background', relative to the player.

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

Multiplayer Game

Overview

The local multiplayer of the game is the real meat of the game, by design. The single-player in fact largely consists of matches against artificial intelligences for each character. The gameplay sticks to a simple, arcade-like format for multiplayer – players enter a multiplayer mode, then pick their characters, then pick a background stage (or one is selected randomly), then a match begins.

Future builds could eventually be enhanced to allow for online multiplayer, though significant structural optimization might be necessary to ensure it works.

Max Players

By default, the gameplay is one-on-one. If time allows, the game can be expanded to provide for four-player local as well, but this is deep in the optional objectives list.

Characters

Players select characters as a major aspect of character selection; as each character is planned to have variations on the game rules, and a distinct array of specials, the selections can be one of the most intense moments of a match.

Variant Modes

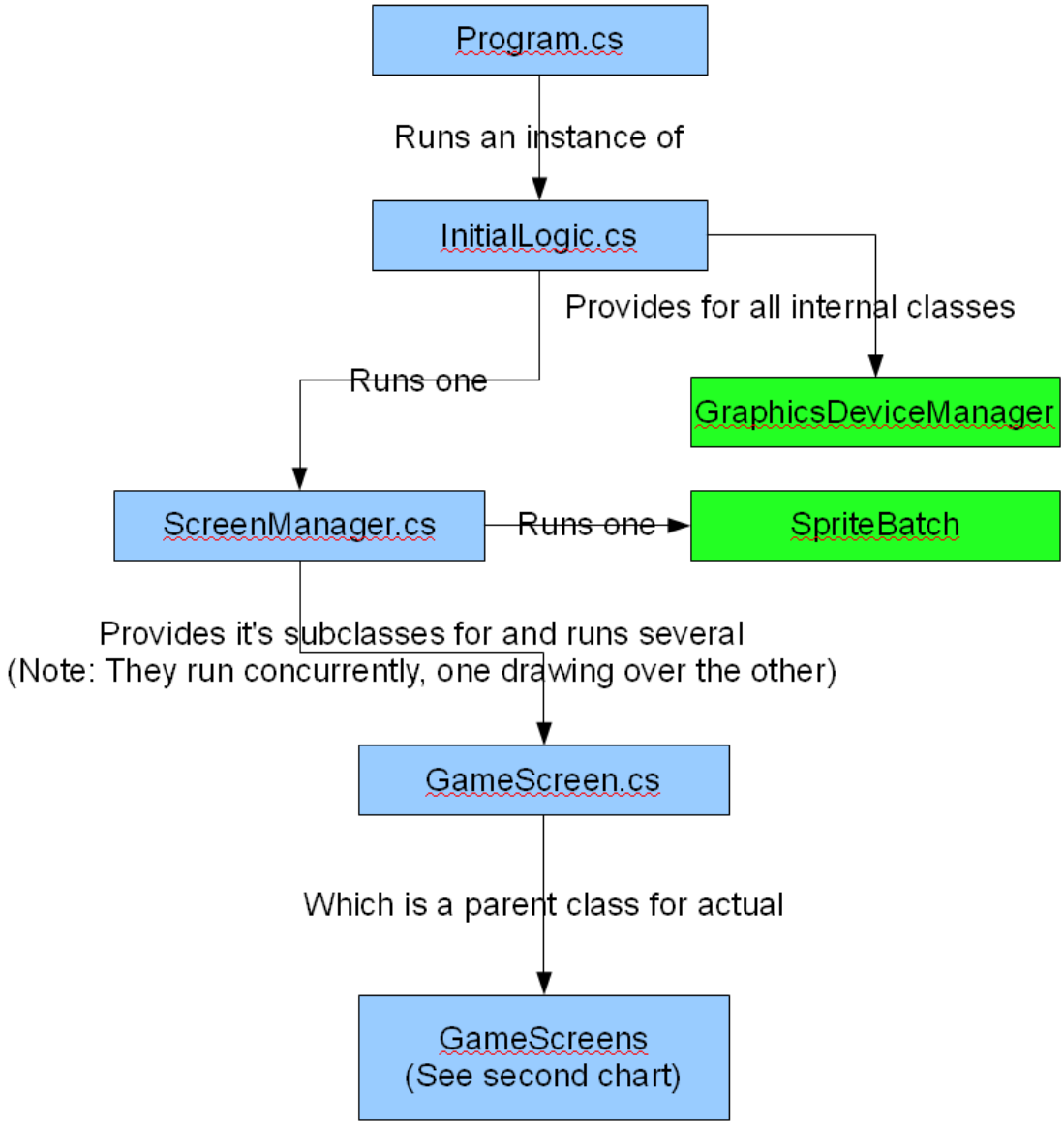
A goal feature for additional implementation is the offering of variant multiplayer modes, which change various game rules to provide a new context. These modes are popular for casual players for more 'party'-type play, and for hardcore to practice various alternative contexts.

Ideas include modes where pieces are twice as large, or half as large, or where combos don't offer any stacking benefits.

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

System Layout

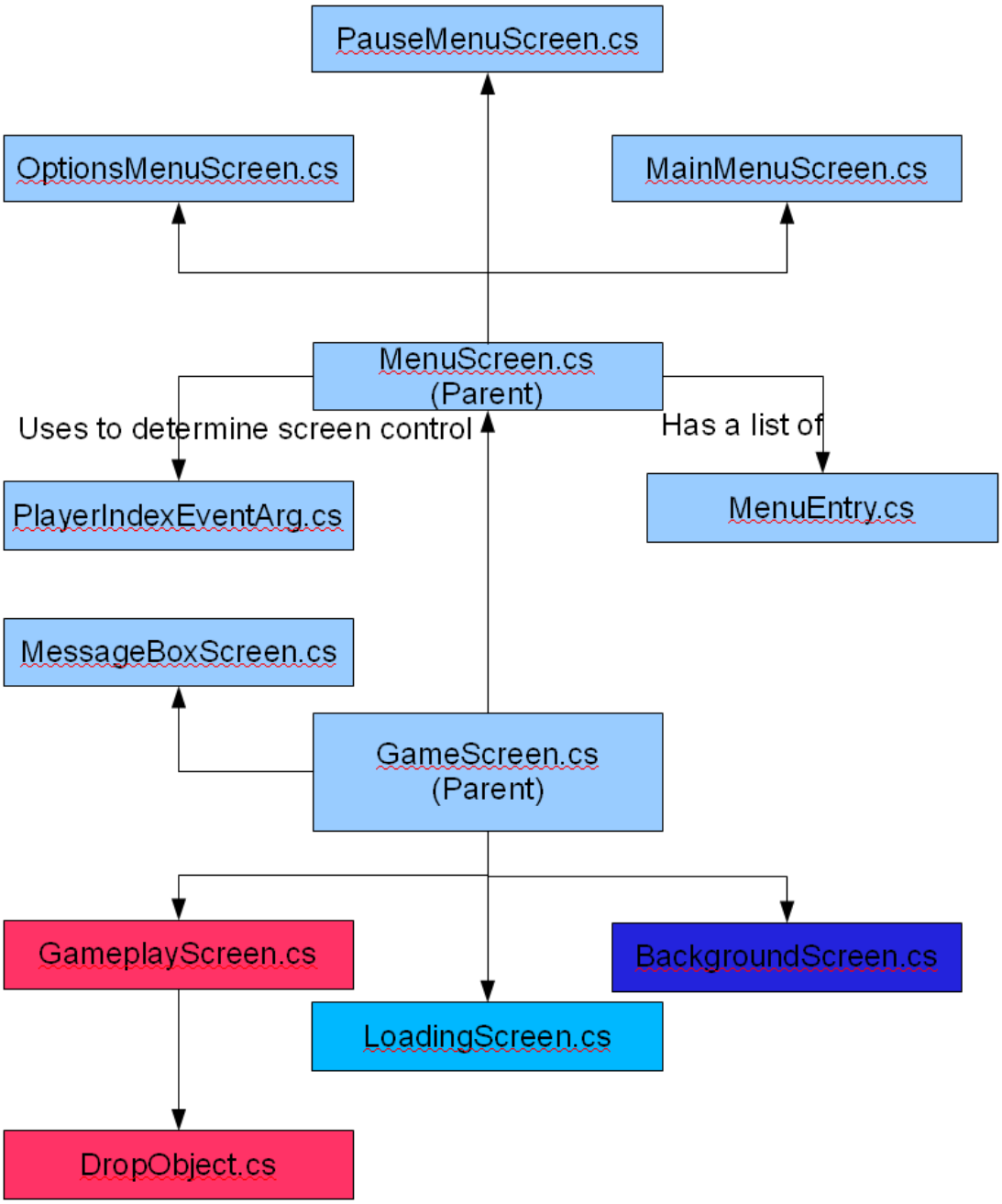
The QuadBoom program design is largely very simple; the actual gameplay structure is implemented in just two classes, with some surrounding structure helping to provide for easy modularization (which will likely be necessary in the implementation of future features).



Blue = found-and-modified, Green = XNA-provided, Red = made
Deeper blue = heavier modification

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

GameScreens Chart



Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

Significant Classes

QuadBoom's class array contains many classes from an XNA ScreenManager sample; only screens I created, heavily modified, or that are *particularly* significant are spoken of here.

ScreenManager.cs: The ScreenManager represents the primary operating core of the entire system. While Program.cs and InitialLogic.cs start some parts, the Screen Manager handles up to (in the current build, and it's unlikely any version of QuadBoom will ever need more) four screens at once. This allows the screens to run in a similar sense to layers, letting them operate largely independently of one another.

It is the primary 'running' class, and contains the active “update” and “draw” methods for the entire system, instructing each screen to run it's “update” in order, then running all the “draws” in order. It handles the insertion and removal of screens as well, and also accounts for instructing screens as to which screen receives controller input.

This class was part of the sample, and largely went unmodified (except for a few small 'put-it-into-format' cases).

GameScreen.cs: The GameScreen type is what ScreenManager carries several of. Pretty much every major code piece I worked on inherits from this class so that the ScreenManager can fully control it.

It provides a few fields and premade functions for handling transitioning a screen on or off, and requires four primary methods be overridden :

LoadContent is run during initialization, for the purpose of loading graphics, sounds, etc. that apply to that screen.

UnloadContent provides a slot for any special commands that need to be run as the screen is removed; in all cases during Design Alpha One, this consisted of a simple “content.Unload();”, which rapidly drops everything.

Update handles all functions to prepare for drawing on each frame; for example, in GameplayScreen, this is where all calculations related to if and how a piece should move *independent of input* are run.

Draw runs after all screens have finished their update, and, per the name, draws all data the screen needs to draw. Note that most screens will *not* fill the entire screen with their drawing; this is intentional, as it allows screens to layer onto one another. For example, the PauseMenu screen places a semi-transparent black square over the screen (using a special code in the ScreenManager) so that the other screens are still visible while going through the menu. (Notably, this introduces a classic 'cheat' common to older Puyo Puyo builds, where a player could pause and take in the entire situation; newer versions hid all pieces when the gameplay screen wasn't active to reduce the effectiveness of this.)

MenuScreen.cs: A subclass of GameScreen, but a parent class for actual menus (the current game has a placeholder Options menu, the Main menu, and a Pause menu). This class runs a simple vertical menu which can be moved up and down through, with “A” as a select button. (Functions for mobile options to move left or right are not in the default implementation, thus not in Design Alpha One, though I did go through the input handling a bit to let it accept alternate 'accept' buttons, essentially just to prove I understood the code sufficiently to do so for later builds.)

PauseMenuScreen.cs: A subclass of MenuScreen, only particularly notable for it's demonstration of the screen-layering that ScreenManager allows – and also for being able to instruct ScreenManager to fully reset the GameplayScreen or return to the main menu – amazingly, this can qualify as a nontrivial function!

BackgroundScreen.cs: Typically the first screen loaded into the ScreenManager, and almost always on the bottom. Modified to display a selected image (in this case, one I quickly crafted using Paint.NET – looks pretty nice, though, huh?), it also is a constant demonstration of the layered draw system, and could easily be coded to take reactions from other gameplay elements as well.

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

GameplayScreen.cs: While a skeleton was already in place from the XNA Sample, the end result is essentially entirely my creation. It is here that all the core game logic that makes QuadBoom into an actual game is implemented. Some GameplayScreen functionality will, in later alphas, need to be shifted to a new class (perhaps **playArea**?)

The screen defines integers to account for piece sizes, but the most significant reused variables include:

playAreaX and *playAreaY*, which define the width and height of the playfield (defaulting to 6*12). Most related numbers (such as how wide a tile is) are almost *never* hardcoded in, instead being calculated off of this; this allows for potential future modifications such as adding more rows or colums.

framesPerMotion defines how rapidly pieces in play drop down – smaller is faster. Currently, this variable is hardcoded to 15, although the default in traditional Puyo Puyo is actually 60 (and drops to as little as 3 in regular play, speeding up over time or as the player goes up against harder opponents).

Two objects of the **DropObject** class (further defined below) represent the two active pieces; these pieces are the ones the player can control at that instant. When they land, they are merged into the playfield, and two new ones generated at the top of the screen.

Up to *playAreaX* * *playAreaY* objects of the same class make up the *playField* two-dimensional array (*NOT* an array of arrays, as is common in Java – the two are different in C#, and the Xbox 360 is apparently faster at handling 2D arrays). This array starts off empty – whenever the 'active' pieces stop moving, they are written into the array before being replaced.

The GameplayScreen code also carries a group of texturs for the playfield frames and actual drops. (DropObjects receive these icons from the playscreen so that only one of each needs to be loaded, no matter how many are in play.)

Major methods in the GameplayScreen include:

EliminationLogic is run each time new pieces are written into the Array, testing to see if any groups of four pieces of the same color exist (using a recursive *EliminationSpider* method, starting from the upper-leftmost piece of each potential group, and a method-specific array that indexes each group to one value, then checks the frequency of that value). Note that the method has to run again if any pieces are eliminated, to make sure that any possible chains are accounted for.

Actual pieces are removed using the *DestroyPiecesOfState* method; **DropObject** items have the group index written into them during *EliminationLogic*, then reset at the end of it. If *DestroyPiecesOfState* is run during the *EliminationLogic*, though, it neatly removes targeted groups. (Testing verified that groups of many different sizes, as large as eleven, were properly eliminated using the method.) This method has a *CalcDrops* submethod, which makes sure that any pieces above removed pieces are dropped to a new location. (In the current build, they fall immediately, rather than in the coded-in-but-unused smooth motion, though, as the game itself lacks finer states than 'pieces are not moving' or 'pieces are moving').

ProcessMotion runs during the *Update* method, decrementing a counter to determine if the active pieces need to move, then moving them if they do – and marking if they've stopped, which indicates that the game now needs to run *EliminationLogic*.

GenerateNewPiece takes a target location as input (needed because the pieces draw based on their location), and returns a piece of a random element.

Update, Draw, and HandleInput all work like the versions in other classes.

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

DropObject.cs: The core of gameplay centers around manipulating “Drops” - thus, they're given their own unique subclass. While a variety of control methods for them, along with their own *subUpdate* and *Draw* methods (called in the gameplayScreen's respective methods), are provided, the most interesting bits are:

The *DropElement* enumeration, which marks what 'color' (Air, Fire, Lightning, Water, Wood, or Trash) a piece is. (Notably, “Trash” doesn't show up in normal play, and has its own handling rules.)

The *MotionState* enumeration handles what the piece is doing right now – the most important ones being *Still*, *Falling*, *Active1*, and *Active2* – the last two representing that these are pieces being manipulated by the player at this time.

Although the array itself can be used to store the pieces, each piece also stores its own X and Y locations, to make running many commands easier. The only time that direct synchronization is necessary is if pieces are dropped due to pieces below them being eliminated, and when the pieces stop moving after being in an *Active* state.

RecalculatePositions and *CalculateTargetPosition* take the piece's current location on the field, and calculate the upper-left corner of the drawing rectangle based on them.

ShiftPieceLeft/Right and *StepPiece* handle moving the piece around the board, typically in response to other commands.

StopPiece is called to set the piece as no longer active; notably, the piece can return to being active after stopping (due usually to pieces below it being removed from play), but this method will only be called once – as the piece is auto-stopped at the end of a fall during *subUpdate*.

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

Implementation Schedules and Estimates Measured Against the End Reality (circa 1.50)

Prototyping methodology

This project will be run in a series of prototypes – each seeking to complete one significant layer of the functionality in a complete form, then move on to the next. This prototype-to-prototype methodology represents, to me, the most important and critical piece of Extreme Programming practices.

While I underestimated the scale of each individual prototype, sticking firmly to "one feature working at a time" kept the project running in small, manageable bites, fixing one glitch after another.

Timing Estimates

I estimated the time per feature as follows:

** Requirements/Design: 10 hours*
** Implementation (Menu system): 15 hours*
** Implementation (Gameplay core): 25 hours*
** Implementation (Special Moves system and character variations): 15 hours*
** Testing/Bugfixing: Up to 60 hours – as much as 20 after each Implementation phase*
** Write-Up: Up to 10 hours*
** Presentation: Up to 10 hours to design/script/record gameplay videos, Up to 3 for practice.*

Requirements and Design, the write-up, and the presentation were all about right – the Presentation took me around 6 to study my notes for, and 3 for practice, and this write-up took around ten hours to re-do. Implementation of the Menu system was trivialized down to 5 hours by my finding the XNA ScreenManager sample – all I had to do was adapt it, make sure I understood it, and spend perhaps half an hour testing it thoroughly.

This left the Gameplay Core – where I found myself much more heavily challenged, and where my estimate ceased to be accurate. Technically, implementation in terms of actual code writing took on the order of 30 hours; however, testing/bugfixing (including one persistent glitch that I never did successfully break past) went well over the intended 20 hours for the phase – typically, I wound up spending two hours testing and figuring out how to bugfix for each hour of actual concepting and coding.

It can't help that what I called the "gameplay core" here was actually *three* of the four phases in my Gantt Chart – and that I only got through the first of them in the end, taking as much time as I had put for the *entire* Gameplay Core for just that.

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

XNA Game Studio Challenges

Microsoft's XNA Game Studio will be the development environment for this game; this is useful because even games implemented for Windows using XNA can be easily ported to the Xbox 360, and it provides easy access to many of the Microsoft DirectX libraries. It is regularly used by many studios – in particular, EA's Pandemic Studios often used it for producing show demos.

However, like all Microsoft products, I have to expect to have to deal with strange, unexpected circumstances. My timing estimates may have to go up as much as 5 hours for each implementation phase solely for accounting for figuring out specific quirks of XNA's DirectX support.

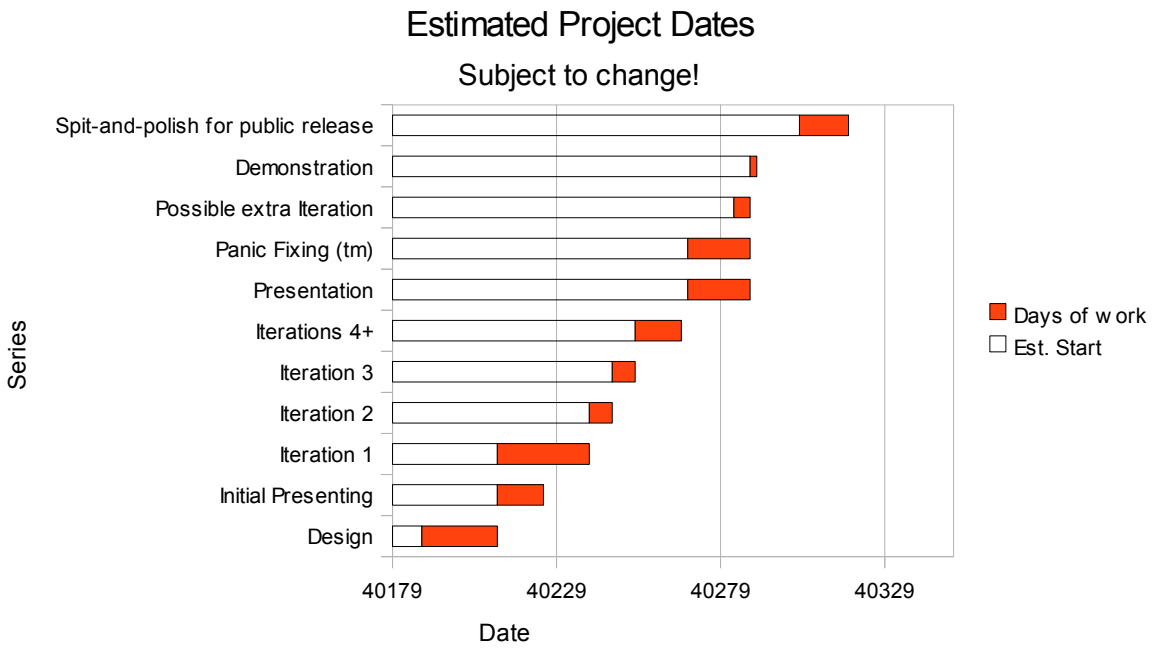
The Visual C# Environment (as part of Visual Studio 2008) was not a problem, with excellent tools for error-tracing. However, a more significant issue came from dealing with glitches in the XNA Studio runtimes – including a persistent glitch where the software would define two completely unrelated positions on the screen with the exact same coordinates in the exact same screen. Not fun.

As I learned how to use XNA features and work with the code I'd already acquired, development grew easier over time, and (outside of the above-mentioned glitch) I was successfully tracing my way through coding difficulties with fewer issues over time, until Design Alpha One was complete (except for it's fail condition check).

Copyright © 2010 Erik Ottosen – All rights reserved – Run away!
Original game © Compile/SEGA 1990-2009

Initial vs. Final Gantt Chart

Here is the Gantt Chart that I presented during the introduction phase of the project:



Naturally, with only Iteration 1 completed due to underestimating its scale and the kind of time I would have to seriously work on the project, I essentially lost three entire phases of the project – however, I did complete Iteration on time with the end of the initially-planned Iterations 4+, and kept largely in time for presentation, panic fixing, and demonstration.

I do still intend to work on the later iterations over time to develop the game into a releasable form, as well.

The most important thing I learned as a result of this? *Never rely on any but the most conservative estimates of available time.* Related, there's a reason why teamwork is the norm in these sorts of projects in the real world – even just two people is a massive boon for end time efficiency, comparing my time developing projects last semester to this.