Garrett Gu
Mr. Hudson
Computer Science 1 PAP/IB
20 May 2016

The Tau Manual

Tau (τ) is a new two-player Atari Pong™-like game that substitutes the traditional rectangular arena for a circular one.

Instructions:
Before starting a game, use number keys to indicate how many games are required to win each round.

Click the circular play button to start a game.

Player 1 controls his/her paddle with WASD keys.
Player 2 controls his/her paddle with directional keys.

	Player 1	Player 2
Clockwise	W or D	↓ or →
Counter-clockwise	S or A	↑ or ←

Survey results indicate that using up/down and W/D arrows allows for more natural movement.

The goal is to keep the ball from hitting your side of the arena by colliding the ball with your paddle.

The location of where the ball collides with the paddle also determines the trajectory, leading to strategical moves, such as keeping the ball on your side or aiming the ball towards the edge of your opponent's side.

Power-ups and power-downs are available to jeopardize results. Power-ups are collected by hitting the ball into them. Some power-ups only apply to the player who most recently touched the ball. Powerups wear off after 10 seconds.

A comprehensive list of power-ups is on the next page.

Power-ups (colors are blue-shifted to enhance visibility):

Paddle size-up: increases paddle size.

Paddle size-down: decreases paddle size.

Ball size-up: increases ball size.

Ghost: ball regularly transitions between transparent and opaque.

Lightning: increases ball speed.

Snail: decreases ball speed.

Warp: ball regularly transitions between fast and slow states.

Wobble: ball wobbles.

Arrow: redirects ball in the direction the arrow currently points at.

Random: applies a random power-up, excluding “Arrow” and itself.

A Short Discussion over the Development Process of Tau

Tau is a two-player ball game that's similar to the classic Atari Pong™ game, but brings a completely new and original twist to the tried-and-true classic. The arena wraps into a circle. Each player's goal is to keep the ball from falling out of the circle on their side of the screen by rapidly moving their paddle around the arena to collide with the ball.

I think that I learned a lot when making this game. My prior projects primarily involved Java, but this is the first major project I made entirely in C++. One major difference between the two is the use of separate header and source files. I learned how to prototype all classes and functions. Another major difference between the two is automatic vs. manual garbage collection. In C++, objects must be deleted manually to avoid resource leaks. Although my program still leaks somewhat, I think that the resource leak is definitely not as bad as it would have been if I had not paid attention to memory. Another difference between the two is both pointers and references in C++ as opposed to "references" in Java, which are more similar to pointers in C/C++. I decided to use pointers everywhere so I could stick to the Java-like ideas. My program also uses multiple inheritance, which does not exist in Java. Also, C++'s random function is not automatically seeded like in Java, which caused a few problems with randomization in multi-threaded applications using the thread-safe qrand function, which are further described later in this paper.

There were many reasons why I chose Qt to write Tau. Primarily, Qt provides a far superior graphics system compared to WinBGI. Qt's robust graphics system allows

full RGB control over colors, enabling an incredible 16 million colors compared to WinBGI's 16 colors, allowing cool visual effects such as color transition and gradients. It also enables transparency, a large part of most of my animations and transitions. Its simple-to-use yet complex-in-practice affine transformation capabilities allows me to easily and relatively cheaply scale, rotate, and otherwise transform my graphics items. Also, it allows easy embedding of images, which greatly simplified drawing some objects such as power-ups. Last but not least, its amazingly easy-to-use anti-aliasing capabilities dramatically increases the sharpness of text and graphics. In addition to its remarkable graphics prowess, Qt also handles mouse input reasonably well. Compared to Qt, WinBGI handles mouse input with the grace of a giraffe on a pogo stick. For example, Qt can still find the mouse position even when the window is moved, but WinBGI's mouse detection relies on the mouse's position on the entire screen. I briefly considered WinBGIm. However, the large disparity in colors prompted me to choose Qt in the end. Also, WinBGI is very old, and was probably written with C in mind, and does not utilize the object-oriented features of C++ to the fullest, unlike Qt, where most library items are classes. Last but not least, BGI was designed with DOS-based systems in mind, so its portability is limited. Qt, however, was written with portability in mind, and is currently compatible with Windows, multiple variants of Linux, OS X, as well as mobile platforms such as Android, iOS, and Windows Phone. Please note, however, that porting to mobile platforms requires a small amount of additional programming to accomplish.

Not only did I learn a lot about C++ concepts, I also learned a lot about Qt programming. Many parts of my program rely on Qt-specific concepts, such as signals and slots. Signals and slots in Qt would most likely replace listeners in vanilla C++ code. Signals in Qt act as functions, but the programmer does not define their bodies. Slots in Qt are pretty much identical to functions except for definition. In Qt, a signal can be connected to a slot, and any time the signal is called, its connected slots can detect the signal and will execute. Another nuance in Qt programming is the use of QTimers. QTimers are different from simple looped threads because they do not require separate threads. Events are simply handled when control over the processor is returned to the main event loop defined in the main function. Therefore, even though Qt can only handle a small number of threads, the allowed use of QTimers keeps the number of threads in the program to a minimum. The difficulty of terminating QTimers during execution prompted me to use threads for animations and transitions instead. Since these threads usually execute only for a split second, I don't expect the maximum number of threads to be exceeded easily. Another interesting fact about threads in Qt is that each thread possesses a separate randomization seed. Even though the random function in Qt is thread-safe, it must be seeded in each thread before it can be used reliably in that thread. This caused many problems pertaining to the spinning arrow around the ball that appears before the ball begins to move. Since the arrow is considered an animation, its motion is contained in its own thread. However, since a new thread is created every time the ball returns to the starting point, the supposed randomization of the arrow's final angle was not random at all, and was theoretically

equivalent to randomization with srand(1). I overcame this issue by seeding the random function every time the ball's initialization code was called with the current time in milliseconds.

I met a lot of difficulties when writing Tau. The first difficulty was setting up the IDE/SDK required to write code using the Qt library. Eventually, I decided to write code on my own laptop which I started bringing to school every day. Another major difficulty was allowing the program to listen to multiple simultaneous key holds without problems. I wanted the two players to be able to act separately and move in a constant speed. Initially, the players would start off slowly and then speed up as the key is held for a longer period of time. Also, one player's paddle would stop when the other player pressed his button. This is due to the fact that my detection of key holds relied on the operating system's keyboard auto-repeat. I ended up fixing this problem by using the Qt library functions to determine when the keys were due to auto-repeat, and I opened two threads that were started every time their corresponding player's key was pressed, and were stopped every time their corresponding player's key was released. When a movement thread is moving, the player's paddle moves in a constant speed. However, the act of creating and discarding new timers every time the player started and stopped moving created another problem by introducing an unbearable lag on Windows that seemed to worsen as the program kept running. Therefore, I eventually replaced the movement timer mechanism with a flag-based system that delegated the paddle's movement to the frame refresh timer. Every time the player held down a key, a flag would be set, which caused the refresh timer to move the paddle at each frame, and

every time the player released that key, the flag is set back to the idle position, thus stopping the motion of the paddle caused by the periodic refresh function. Another major difficulty was the collision between the ball and the paddle. How would I make the collisions seem realistic while maintaining a player's intuitive control over the ball's direction? I decided to make the ball's trajectory depend on both the slope of the line segment it makes with the center as well as the position of the ball on the paddle relative to the size of the paddle. One of the largest difficulties, however, was running the program in Windows. Microsoft Visual C++ would not treat timers the way I expect because it's timers were often wildly inaccurate and slow. This issue is far less serious on MingW for Windows compilation. There are still some (very) occasional lags to be expected, but I fixed many of the crashes and other bugs that affected my program when it was first ported to Windows. As of version 1.032, the problem with lag in Windows has pretty much been fixed. Interestingly, porting to Windows did not cause the bugs, it simply made the bugs visible.

Tau was written with an object-oriented focus in mind. The Arena class manages the circular arena as well as the background gradient. The main arena object is the only object that is kept during the transition from the main screen to the game scene. The Ball class is a graphics item that represents the ball. A constants file contains my constants and is used throughout the program for metrics such as window width, height, timing, etc. The GGameScene class is an extension of QGraphicsScene and is used for the main game screen in general. The GGraphicsView class extends QGraphicsView and contains the current scene and manages keyboard and mouse input. The

GMainMenuScene is similar to GGameScene but manages the main menu instead of the game screen. The helper file contains functions that are useful such as a randomInBound function and a function to find the acute difference between two angles (more complex than it sounds, since the angles in question could be outside of the range $0° \leq \theta \leq 360°$). I also created a function to find the pythagorean distance between two coordinates. The playButton class extends QGraphicsItem and represents the play button on the main menu.The Player class is a Graphics Item that represents the player's paddle. The powerUp class is a Graphics Item that represents a single power-up. (Power-ups are contained in a QList within GGraphicsScene.) The ScoreDisplay class is technically part of the background and shows the number of points each player has accumulated. The Settings class loads and saves settings to file and currently contains just one metric, the winning score. A pointer to a single Settings object is distributed to each object at runtime. The titleText is a Graphics Item that can actually represent any text that I display on the screen. I chose an object-oriented programming style because it is what I'm accustomed to from Java, and I find that it helps me stay organized by creating a class hierarchy. During initial development, I found that creating getters and setters for each member variable was too much of a hassle and thus created a few public variables (bad). However, I later found that Qt Creator has a button that creates getters and setters for a variable automatically, and attempted to migrate some of my code to the better style.

There were a few features that I planned out but couldn't achieve because of lack of time. For example, I didn't create an MLG mode, >30 power-ups (there are 10), FFT

beat detection, mobile release, AI mode, etc. MLG (major league gaming) mode was a planned skin/audio replacement for the program that hopefully would have opened the doors for more skinning and audio customization. The >30 power-ups goal is of dubious benefit, since the current 10 power-ups are already becoming confusing for users to figure out. The arena in the game pulses according to the given BPM (beats per minute) of each song. I hope to extend this functionality so that a user can input a music file and have the arena automatically pulse according to the music. This should be doable with an FFT (fast fourier transform) over the audio file to divide the audio into its constituent frequencies, and analyzing the lowest frequency, which usually represents the beat in most songs, to find the peaks in audio energy, which are then fed into an analyzer function that estimates the BPM. A mobile release should be a relatively simple goal due to the portability of Qt programs. I believe that an AI mode will be possible by simply extending the current trajectory of the ball until it intersects with the arena, and then setting that point as the target location of the AI. The AI can move at a constant speed towards the target location until it reaches it, and that constant speed can vary according to the difficulty setting of the AI. When playing against an AI like this, “outsmarting” the AI is achieved by launching the ball at a steep enough angle such that the speed at which the ball’s angle with the center changes is greater than the AI’s maximum speed, and the AI thus cannot catch up with the ball quickly enough. The above features might have made it into the current release if I had more time, and will hopefully be part of future releases along with other features and bug fixes.

I had a few difficulties managing my team. Colin was having trouble understanding C++ OOP and Qt concepts such as signals and slots, so I asked him to do non-coder jobs such as creating icons and finding music. When he had problems with doing those, I asked Mr. Hudson to advise, and he sent Colin to help other people with their projects. Therefore, I ended up doing the majority of the work. Fortunately, having a single developer greatly reduced the number of conflicts that occur in git commits, and also increased the overall coherency of the program in my opinion.

I think that I coped with the deadline pretty well. I worked consistently, as evidenced by my daily commit streak on Github. When the deadline approached and I realized that I would not be able to achieve everything in my project design, I did what any self-respecting programmer would do and asked my supervisor for a cutdown on specs. As my own supervisor, I updated the specs quickly and without fuss, which is almost unprecedented in the industry.

Conclusively, writing Tau was a new experience for me. The complexity of this project was a step up from my two previous major projects, Waterfall and Labrat, as indicated by the significantly larger number of commits involved during development. During development, I learned a lot about C++ programming and writing portable application with Qt. Even though I did not reach the high standards that I set out in my game design, I believe that future improvements to Tau will add more features and increase the value of the entire game.