Luca Bruni

Mr. O’Connor

TEJ3M1

30 January 2017

PART 1: RESEARCH

Identification: My culminating task will revolve strictly around computer programming. The game I am going to create is a 1vs.1-type, 2-dimensional shooter. The game does not require any ‘physical’ parts, but there are still some requirements in order for it to function correctly.

The game’s use: This game in particular has no physical use. However, by repeatedly playing the game, players may find enjoyment, entertainment, and use it as a time-killer or a distraction from stress. Otherwise, the game may develop human skills for real-life use such as reaction time, intellect in terms of strategic thinking, and patience.

Commercial Potential: Although I believe there would be no such commercial potential for the game or anything similar using Turing, I believe there may be some potential if other languages were used such as Java. For example, networking may be used in future development of the game so online play is allowed. This of course opens doors for the game to allow micro-transactions to take over. This purchase potential may involve ‘skins’ or camo for players, or pay-to-win aspects such as more powerful weapons at a cost. Also, the game’s rights may be sold to websites for free use by players.

Knowledge/Skills Required: To be able to recreate such a game, there are various requirements to be able to be effective in doing so. Firstly, you will need knowledge of programming theory, especially in a language such as Turing. This may involve knowing aspects such as how the code is read from top-to-bottom, how procedures need to be called, and variables need specific data types in order to function correctly. Not only do you need knowledge of the theory, but also basic knowledge of how to perform certain functions within the language. This is in order to effectively recreate the game with the same features. Some advanced knowledge may be required for array systems and graphics. Along with these requirements, a skill you would need is patience. Programming a game always comes with mistakes, and problems with the program that cause something to not work properly. This is accompanied by a lack of reasoning as to why this may not be working. I can say from personal experience that patience is a need to be able to effectively recreate this game.

Do I have the skills: For most of the program, I have such skills and knowledge in order to create this. This includes knowledge with variables, procedures, loops, arrays, and graphics. I believe that I am a relatively patient person with a lack of frustration needed to complete the project effectively. These personal traits allow me to focus more on learning and progressing with my code as opposed to spending time in frustration. While I do have basic, advanced and theoretical knowledge with Turing, I also have to learn some new things online. This includes a counter-system to avoid global delays. The system will be learned due to the global delays slowing down code unnecessarily. I also have to learn a system for cooldowns on the weapons that I incorporate into my program. A larger cooldown will be used for a more powerful weapon.

PART 2: PLANNING

Materials necessary: There are no physical requirements for this project other than a working computer. - A functional computer with adequate RAM and CPU usage - A version of ‘Turing’ with features required to complete the game - A USB key for the transferring of the game unto different computers

All of these materials should be free unless a USB key is not available.

Schedule:

**Project Received: January 9th**

**Game Completion Goal: January 22nd**

**Report Write-Up Completion Goal: January 28th**

Week 1 - 9th-16th: Write most of the research and planning portion of the project on personal time. Work on the code will be done during class time. This work for week one involves setting basic variables that are a must-have, such as arrays for a standard player model and the bullet movement. This also includes programming a start-up menu and a background for the game. Week 1 should end with the movement mechanics and bullet movement mechanics set in stone. Another aspect of the game that should be completed by the end of week 1 is a hitbox system to check if the ‘x’ and ‘y’ values of the bullet come into contact with the ‘x’ and ‘y’ value of each player. This system will allow for me to continue programming effectively into week 2.

Week 2 - 16th – 22nd (+ extra class days): Retrace the beginning steps I have made into the project and incorporate this into the Building and Prototyping area of the report. Once this is complete, review the mechanics of the game and seek out any small fixable bugs and rework any necessary movement features. Using the hitbox system, develop a health system for the players, as well as a graphic for the health remaining. Graphics should also include cooldowns on each weapon that should be completed by the end of this week. The master loop that runs each procedure should be working and a functional game should be up and running by the end of the 22nd. The class days remaining before other exams should be used for CPU usage optimization, such as only drawing shapes and text once to avoid any lag within the game. Fonts and other aesthetics may be added in the game during this time to finalize it. By the end of the week, the core program should be complete as well as most of the report.

Week 3 - 25th – 30th: This is a time where no core features of the game are to be added, because by then they should be added into the game. This short period of time should be used to add final finishing touches to smoothen the game, and add any possible optimization within the game to allow for the best player experience. Any final touches will be made to the report to ensure a lack of spelling, grammatical and layout issues. The final two days, the 29th and the 30th should be used primarily for bug-testing. This testing should stretch every variable to its limits to find any possible errors and unwanted glitches within the code. Should an error be found, proper action to fix it such as any value changing should take place immediately. Once this testing is complete, perhaps load the project onto a USB, as well as zip the folder and e-mail it to myself in case a technical issue prevails.

Program interaction: Although there is no interaction with the program and any physical circuits, I may explain how the program interacts with the mouse and keyboard, as well as the person using them. To begin, a start-menu will display upon running the program. This menu is interacted with by using the mouse. Clickable buttons will be displayed to the person viewing the monitor. The program allows for the finding of the ‘x’ and ‘y’ values of the mouse so that movements on the screen correspond to movement of the mouse. Clicking the left mouse button on a button with run a different scenario, such as instruction, controls and/or actually starting the game. This is done with Boolean statements and procedures within a loop. Starting the game, various keyboard interactions are to be made to connect with the program. Using keys such as the up, down, left, and right arrows, as well as the standard W-A-S-D-SPACE gaming keys are available in this program, where each key performs a different task. For the player viewing the game, everything is organized into simple graphics that help the player understand what is going on.

PART 3: BUILDING/PROTOTYPING

Steps taken to complete – Documentation in ‘point-form’:

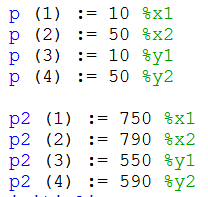
**Screen**

* Set comfortable parameters of the screen’s ‘x’ and ‘y’ boundaries. This is done first so any future graphical code is correspondent to the screen’s size.
* Remove any unnecessary buttons or scroll bars. This step is optional, although is most likely needed for accurate parameters of the screen as well as an aesthetic game window.
* Set the view as “offscreenonly”. When the View.Set ("offscreenonly") command is given, Turing no longer draws to the onscreen window when any drawing command is given. However, it does update the offscreen window. When the View.Update command is given, the entire offscreen window is copied to the onscreen window. These commands also avoid flickering, and ensure properly drawn shapes or textF:\#gamefolder\a60de0a316d5f7afa4e90d308a966dfb.png

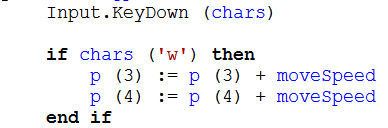
**Player Model**

* Create the player models by using an array. Using a square, you will need a 1x4 array. These four locations will be the ‘x’, ‘y’, x2’, and ‘y2’ locations for the square. Repeat this for a second player in order to create two player models.

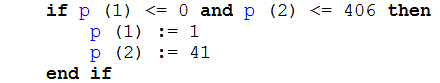
F:\#gamefolder\2222222.png

****

* Create a way for these models to move on the screen. You can do this by creating a variable by the data type ‘array char of boolean’, followed by the Input.KeyDown (variable) command. Not only does this create the key variable, but it reads whether or not the key is pressed down or not. These commands are followed by a series of ‘if’ conditions that state if the certain key is pressed down, then an ‘x’ or ‘y’ value of the player model moves in correspondence.



* Make sure the players cannot move off of the screen or in a location where they should not. Using each of the array’s ‘x’ and ‘y’ values, state ‘if’ conditions, where the ‘x’ location cannot be greater than the maximum ‘x’ value of the screen or less than zero. Make sure the ‘y’ cannot cross into the other team’s zone or go past the maximum ‘y’, or zero.

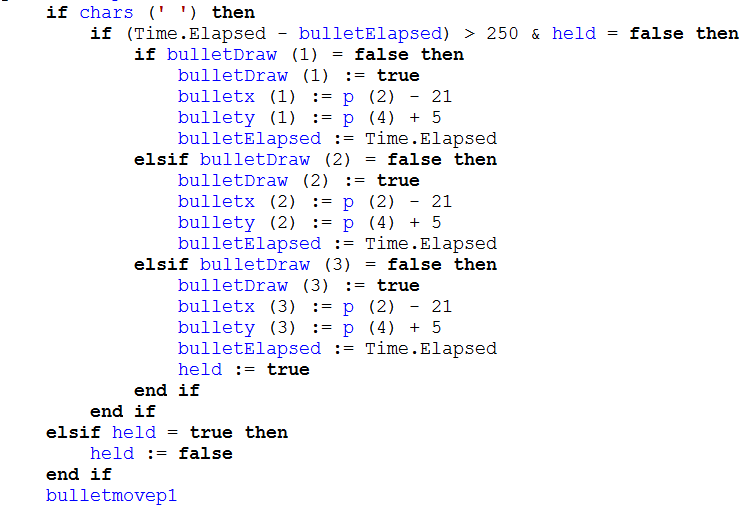


**Bullets**

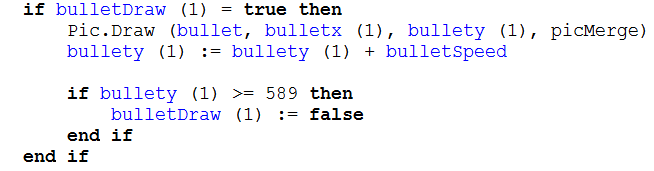
* Create a model of the bullet. This can be done using a 1x4 array but I used a premade image in the .bmp format. The next step is drawing the bullet. You can do this by creating a variable using the Pic.New command, then using Pic.Draw to transfer this variable onto the screen graphically.

F:\#gamefolder\6666.png

* Movement of the bullet could be done using ‘if’ and ‘boolean’ statements. Firstly, you need to create a key that when pressed, the bullet “shoots”. Again, this could be done using that key variable you made, along with an ‘if’ condition stating that when the key is pressed, the bullet will move in response.



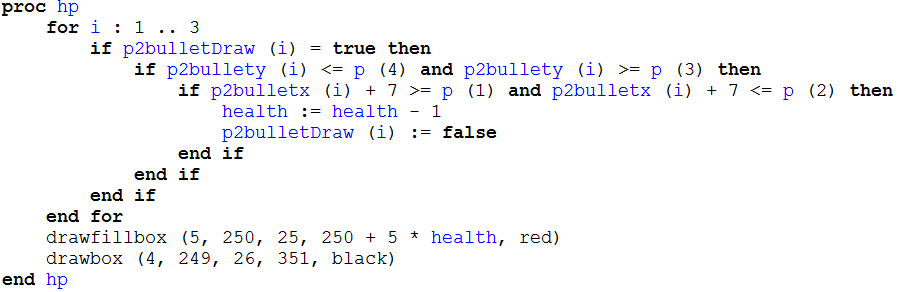
* Boundaries of the bullets may be set using a series of if conditions. First, check if the bullet has struck another player. This can be done by comparing the bullet’s ‘x’ and ‘y’ values with the ‘x’ and ‘y’ values of the player arrays. By doing this, you are completing a proper collision check. Once this checking has been done, you can use another ‘if’ condition that you can use later for the health system based on this action. Secondly, you must check if the bullet reached the end of the screen without touching the opposing player. You can do this with yet another ‘if’ condition. The statement will check if the bullet’s ‘y’ value corresponds with the ‘y’ boundary set for the screen. If so, the bullet disappears.



* Mentioned earlier, you must have a Boolean variable in order for this to work. A Boolean value that states if the bullet is drawn can be set to either ‘true’ or ‘false’. For example, if the bullet hits the player or the boundary, the variable may be turned ‘false’ so the graphic “disappears” on screen. This opens up an opportunity to draw another bullet in its original location – just above YOUR player model, and NOT where it was just erased from. This may be done by setting the variable back to ‘true’ once the shoot button is pressed again.

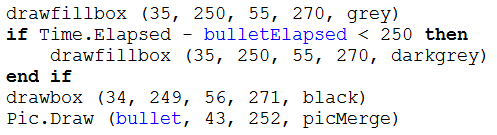
**Health System**

* This system is a must-have for the game. This allows for the game to check whether or not the bullet has hit the player and if so, to take away a certain amount of their health.
* Firstly, check if the drawn bullet variable is set to ‘true’. This allows for the game to check if there is in fact a bullet on the screen.
* State an ‘if’ condition where if the player is hit by the bullet, the bullet draw variable is now ‘false’ so it disappears, as mentioned earlier.
* This is where you must make health variables for each player: they should be an integer!
* Within the ‘if’ condition, take away health from the player *taking* the hit from the bullet should the bullet come in contact with their ‘x’ and ‘y’ locations.
* You may display each player’s health in the form of a number or a health bar.
* If you choose to display the health as a bar, the ‘y’ value must include the health variable during multiplication. This is so that whenever the health is reduced, so is the ‘y’ value of the health bar, giving you the graphic of how much your health is reduced each hit.



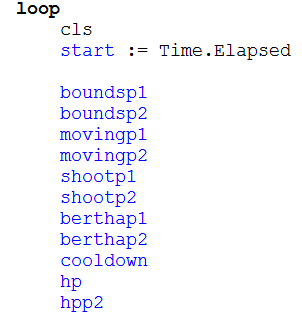
**Cooldown System**

* This is yet another system in the program that is important. When shooting a bullet, you need to know how long it is until you may shoot that same bullet again.
* To do this is simple. When you have a bullet available, you may display a graphic with the bullet inside a light box. When this bullet is not available, the box can turn darker, indicating that you may not use it just yet.
* This can be done using a timer. Time.Elapsed is very important as it takes the time allotted from when an action is performed all the way until the present time. For example, you may create a variable that is an integer equal to zero. Then, when you press the button to shoot the bullet, you may make that variable equal to Time.Elapsed. The program will keep track of how long it has been since you pressed the button.
* Following this, you may set an ‘if’ condition. So, you can make the cooldown time as you wish, as it takes count in milliseconds. If the variable under Time.Elapsed is less than a certain time (for example, 1000 milliseconds or 1 second) then you may draw a darker box around the cooldown indicator. If anything else happens, then the box may return to a lighter colour, indicating that the cooldown is over.



**Game Loop**

* The bold titles above reflect on each procedure(s) to use in the program. Now, it is time to call onto those procedures into a loop, so the procedures are repeated until otherwise stated. First you must make the loop, and clear the screen so that when you View.Update at the end, everything you have drawn will be drawn real-time and nothing will flash. Now, call on the procedures simply by typing the name of the procedures within the loop. Be careful when doing this, as the loop must be placed after every variable has been created so the loop understands that those variables exist. After all these procedures are done, follow up with the View.Update command. In this loop, you may set a command so that when a condition is met, you can exit the loop entirely. For example, if a player’s health reaches zero, you may exit and state that the opposing player won, effectively ending the game and giving an opportunity to begin a new game.



Changes made as I tested the program: There are many changes I made to the program along the way. One such example would be drawing the bullet image for player 2. Since this is a two- dimensional game, one bullet must go up towards the player 2 from player 1, and another bullet must go down towards the player 1 from player 2. One issue I had is that when drawing the image, the bullets would be facing up no matter which direction they were shot in. For example, the bullet shot from Player 2 (from top to bottom) still faces Player 2, which makes absolutely no sense. To fix this, I went back in the code and made another variable for the same picture: except with a rotation. I used Pic.Rotate and rotated the picture 180 degrees for player 2. Another big change I made was from using fixed values for the player model to using an array. An array was so much easier because it helped me actually keep track of which location and side was which when doing collisions as opposed to being confused with a bunch of different numbers at hand.