

Project Documentation for Zombie Trail

Requirements

- Basic Requirements of the Program
 - The program is designed to be a fully playable (the game will not crash, and the end goal of the game is reachable)
 - The program will implement the C++ coding language
 - The program will have 2D graphics done using the Direct X SDK
- Basic Game Requirements
 - The game is to be split off into three individual sections, each to be worked on by a team member.
 - “Exploration” (Gregory)
 - The exploration section is to entail game-creation and initialization as well as the game section.
 - The exploration section must provide the player a means of seeing his current progress and status in the game, as well as give a visual cue to the location the player is currently at.
 - The exploration section may feature multiple paths to take (that subtly alter gameplay difficulty) as well as random events.
 - “Camping” (David)
 - The camping section will allow the player to modify in-game resources through game play with a trade off of other resources.
 - The camping scene must display an image of the camp, as well as show the current status of the game.
 - Decisions made in the camp will be done using a keyboard-navigated menu system.
 - “Battle” (Fredy)
 - The battle system will offer the player decisions via a menu on what his ‘characters’ will do, then the enemy AI will make decisions on what they will do in a turn based manner.
 - The battle system must implement current game data, as well as data it is fed to modify what will be shown in the battle, as well as the difficulty of the battle.
- Case Descriptions
 - **Void initGame()** – initGame function is called at the start of the program, it will set the game status to a predefined state, then begin the game.
 - **Int RandomNumber(int min, int max)** – RandomNumber will use a random generator to create a number within the range of min and max, then return it

- **Int getScore()** – returns the player’s total score. Score is calculated based on the current game status, and is constantly changing. It can go down, as well as up.
- **Int GameOver()** – Called when the game is over, tells the render program to render the Game Over screen, as well as display the player’s last known statuses and score. This effectively ends the game.
- **Void SanityCheck()** – this checks the game’s current data values of the player’s status, if a value goes over the minimum or maximum allowed for that value, it will modify it to the minimum or maximum. It will also check for game over scenarios if applicable.
- **Int WINAPI WinMain(HINSTANCE hinstance, HINSTANCE, hPrevInstance, LPSTR lpCmdLine, int nCmdShow)** – this initializes a windows program, and acts as a normal int main() function for console based code. This program will run initializations, set the maximum allowed frame rate, then run the render programs. Returns a 1 on successful program exit, or -1 on error.
- **Int initD3D(HWND hWnd)** – initializes the direct3D rendered, loading up all applicable textures and APIs and font data.
- **Void explore()** – the render program for the ‘exploration’ section of the game. This will render the current status, as well as the ‘envoy’ (the three cars that represent the player’s team) and the current terrain the player is on. It will also run all necessary calculations when the game ‘moves forward’ to the next X position. Returns -1 on failure, 0 on GameOver.
- **LRESULT CALLBACK WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)** – this is the message handler for the windows program. Though unused in our program, this is still recommended for popup messages and errors that may occur that may not be caught (fatal exceptions, stack overflows, ect.)
- **Void cleanD3D()** - releases all the texture and sprite data from memory.
- **Void loadTexture()** – shortcut function that calls D3DXCreateTextureFromFile(), allowing us to input the minimum necessary for texture loading.
- **Void initDInput(HINSTANCE hInstance, HWND hWnd)** – initializes the DirectInput protocols for this window as per DirectInput8
- **Void cleanDInput()** – unacquires and releases the keyboard from this program’s use. If this is not called, the keyboard will be locked from use in other programs, even if this program is closed.
- **Void CampSite()** – renders the campsite scene and displays the current status of the player, as well as open up keyboard-navigatable menus for the player to use to influence the game.
- **Void wutHappened()** – called by CampSite() when a ‘search’ is performed by the menu. Calculates the results of the search, then returns them to the main render for display to the player.

DESIGN

Sequence Narrative

The program begins in the WinMain() function, here the game values are initialized, memory is allocated, and the window the game is displayed in is created.

InitD3D() and initDInput() are then called, initializing the render engine, and the input engine. After this, the frame rate maximum is set, for the use of our game we render the game at 20 frames per second, this is prevent the game from running to fast, especially on very fast computers which are capable of running the game far beyond 120+ frames per second. It then enters the exploration render loop.

Exploration Render Loop – this loop displays the game status, the envoy, and the current background, which is based on the ‘terrain’ that the player is currently on. The game will automatically, after a randomized amount of time, move forward on the map, generating a new terrain, applying an algorithm to the player’s resources, and then displaying the new information onto the screen for the player to read. If the status morale or health are low, then the number of survivors is decreased as per game rules, if the food is too low, then health and morale are decrease also. If fuel is too low, then speed is set to zero, causing time to run out even faster. And if time or survivors is equal to zero, the game ends. If the player runs into a random battle event, then the game jumps to the battle loop, if the player pushes the letter C on the keyboard, the game jumps to the camping loop.

Camping Render Loop – the camping render loop displays a number of menus for the player to traverse. The first will ask why speed camp they wish to design, Fast, slow, or medium. This will determine how much health and morale is recovered, as well as the time that is lost setting up the camp as per game rules. The player may then use the keyboard to traverse the menus to decide if he wishes to exit the game, or search for four of the game’s possible resources, food, fuel, ammo, or survivors. The player will then decided how many survivors he wishes to send out to search for these resources, and how much time they will look for, knowing full well that the player stands a chance to lose survivors he sends out. This may be repeated until either time runs out, at which point the game ends, or the player choses to leave the camp. If he leaves the camp, he returns to the Exploration Render Loop.

Battle Render Loop – The battle render loop places five groups of zombies, each of differing strenghts onto the screen, then displays the three selected player characters. It will then read the current survivors, and display a ‘health bar’, which is based on a percentage of these survivors. The game will then provide the player a choice of what attack to use, and which zombie group to use it on. It will repeat this for all three characters, after that the zombie characters will take their turn and directly attack the player’s survivors. This

repeats until either all zombies are dead, or all survivors are killed. If all survivors are killed, the game ends, otherwise it returns to the exploration loop.

Discuss Alternatives

Flash vs. Java vs. C++ vs. C# - We had to decide which programming language to choose from in the start, each had their pros and cons.

Flash – Pro – easy to work with graphics, well known, very portable to other systems, used in the industry.

Con – Not technically impressive, long and arduous to work with.

Java – Pro – we know java and how to work with it, easy to work with threading and data structures.

Con – We know java already.

C# - Pro – C# is also used by the industry, and was taught to use beforehand.

Con – Most of our team heavily disliked C#, and it is somewhat difficult to work with.

C++ - Pro – C++ is an industry standard and is used even in modern game programming. It'd also be very useful for us to learn.

Con – C++ has a high learning curve as we have not used it before, especially when interfacing it with a graphics renderer.

In the end, we chose to work with C++, so that we could learn a new language this year, as well as one that is an industry standard.

OpenGL vs DirectX – OpenGL and DirectX both offered different advantages and disadvantages. OpenGL is easier to port, and can be used on multiple systems, while DirectX tends to be limited to Windows systems. However, DirectX offers much better hardware acceleration support, while OpenGL relies on software acceleration. Combined with the fact that most modern day games use DirectX to render their environments, we decided it be best to work with DirectX as a learning experience.

Timber resource – The timber resource was originally planned from the start to be used to build barricades and camps, among other random uses that could be decided in later versions. The barricades were removed at one point (this is covered in the Battle System HP usage) lowering the use of timber to simply being used for camping. We decided by this point, timber was just simply an extraneous resource due to the low cost of camping with lumber and the ability to find such a large amount of timber, so we removed it.

Pushing M to move vs. Automatic Movement – Originally the game used M to move forward to allow the player to determine whether or not he

wanted to camp or not. The team was split on this, with some team members finding it a necessary bit of the game itself, while others found it useless and should simply have automatic movement. After some discussion, we decided that automatic movement would add a small element of being rushed to the game, and it was added in.

Planned Camps vs. Camp Anywhere – The game originally was designed around using planned campsites that would appear every so often, this was changed to become a ‘camp anywhere’ system for debugging purposes. But we found it much easier for the player to have the ability to camp anywhere they wish. This also provided a temptation to the player, as over-camping would decrease their remaining time and ultimately cost them the game, as well as helping them.

Battle System HP use – The battle system underwent many changes before reaching its final version. Originally we planned to have each of the player characters zombie targets and thusly permanently killable. We decided that this was too unfriendly to the player, and that the death of one player character would force most players to restart the game. We then decided to change the battle system so that zombies targeted survivors instead of the player characters, and distance or a barricade would stand before them. With playtesting, we decided that distance factors and the barricade would merely time wasters to ease the battle system, and resulted in just mindlessly pushing the attack button until the battle was won or the zombies were finally able to attack.

We decided to start the battle in a traditional roleplaying-battle system, and players and zombies take turns attacking their respective targets, and players could use limited skills to ease the battle and damage they would take.

TEST

Test Plans and Procedures

The testing for this project varied depending directly on what we were trying to test. Due to the use of Direct X in specific, most of the items in this category we had to test were graphical objects and animations. To do this, we would insert the object into the program, test if the object did appear as it was intended to, as well as in the correct position. For animations, we would have to run constant tests to be sure that the animations not only occurred correctly, but did not suffer from stuttering and remained centered in its position.

For the core game engine functions, testing occurred early in development, as we created the game as a text-based game before we added the graphics in. Here testing followed a more standard method. We would

create functions, then test them, looking specifically for not only a clean, smooth run-through of the function, but also for the correct output.

The most difficult of these to test was the random function generator function. Due to the nature of randomization, we could not predict the numbers, but rather hope that the numbers the system drew were as unpredictable as possible, based upon the seed. We found one interesting error that occurred in two circumstances, one circumstance was due to the reliance on C++'s built in randomizer. It drew the same numbers each and every run, creating a horribly predictable game. The second error occurred when we changed the number generation, and implemented a time based seed, at which point the generation refused to give us numbers within the specified range.

Beyond that, the functions that determined the outcome of random events had to be tested, as well as the functions that determined the outcome of 'searching' in the camp scenario had to be unit tested separately from the main program.

The main core itself was tested as code was added to it, piecewise. Each part that was added in, whether it be a major part such as moving forward in the game's map, or a minor part, such as a small random event, was testing individually as it was added. It was then later retested when other things were added in case of a possible conflict. This help true for all three parts of the game core, the exploration, battle, and camping scenarios.

Test Report

The final program testing was split into different parts and was done in a traditional game-testing method in which a number of players would each run through the game, focusing on a different aspect of it, and doing all things possible to it, including things it was not intended to do. This is done in hopes that any glitches or bugs that are noticeable and viewable in the final product can be seen and fixed.

More in-depth testing, such as testing on number generation and core and graphics engines, were done on a per-addition basis, and were unit tested as functions as they were added. The tests for these aspects result in the following:

Random Number Generation : Passed, seeds correctly and generates pseudo-random numbers.

Auto-movement in Exploration(): Passed, occurs after a random, yet small, amount of time. Correctly increase the x position of player

Exploration() Game Over – Passed, correctly detects when three states have been reached. Either survivors or time is zero or less, resulting in GameOver() being called, or Distance is zero or less, resulting in WinGame() being called.

Campsite() menu selection – Passed, the menus had to correctly accept input from the Up, Down, Left, Right, and Return keys, as well as proceed

through the menu selection in the correct order. The menu should also be responsive.

Campsite() resource searching – Passed, function returns the correct amount of found resources based on an algorithm and applies changes to survivors based upon how many ‘died’ in the search.

Battle() scenario – Passed, the battle scenario had to be tested as a whole, rather than in parts. The battle system correctly takes turns, resets itself upon the end of a battle so it can be called again, and correctly returns the end-state of the battle.