

DANIEL GADD

C ++ ASSIGNMENT II

Lecturer – Dr liam noonan

SIMPLE ASCII CONSOLE GAME

**® INTRODUCTION & GAME OVERVIEW ®**

In this project we will cover some basics of object oriented code and design. During the process we will see good use of objects and their associated variables, their functions, polymorphism and inheritance. Our game has three levels and different types of enemy AI, such as pathfinder and random. I aim to create the building blocks of a huge ASCII C++ object oriented console game. Although we will only complete a certain amount of functionally and testing, this platform can be used at a later stage to demonstrate to future employers my object oriented skills to a high level.

Zombies is a well-known game franchise and in this project we are bring them on the ASCII console platform, CHARBIES. This prototype will provide the user with three levels and increasing difficulty while they try their best to gather supplies/treasure for their camp during the great outbreak of the hash monster zombies.

It’s a black and white world and characters are in danger of been wiped out by a charbie outbreak of flesh eating hash keys. Battle your way through three unique levels and grab all the treasure/supplies you can before you get eating alive.

**® REQUIREMENTS ®**

Our game is required to provide a fun experience for users, through simple story and animated ASCII characters on our console. For my assignment I will be using Windows, Special windows functions and C++. Users will be bale understand they story, play the game while enjoying the game flow. The game will provide three levels and increasing difficulty.

We have viewed and identified two other C++ console games that we can consider our general competition. Their functionality has been viewed and tested. I have decided my main unique algorithm that will be included in my game and not found in their game is a pathfinding algorithm, which allows the enemy object to seek you out. This will give our game an edge

[Pjotsze Console Game C++ Youtube](https://www.youtube.com/watch?v=nDdKIxnEyh0&t=49s)

[Holk von Sehren Console Game C++ Youtube](https://www.youtube.com/watch?v=Y4gU2vttNTo&t=59s)

To produce this level of game flow and entertainment we will need to study the feasibility of particular functionalities such as

* Provide Player input keys timeout using the **\_KBHIT ()** function from the WINDOWS.H library. This gives the game a constant frame rate and all other objects keep moving and updating
* Provide user keys as standard using GETCHAR () function
* Provide instant view of the player on screen by developing a display player function
* Provide two types of AI for moving enemies. One will have to be a path finding algorithm so as the player is being chased. This will exhibit a new unique experience from the two console games shared above earlier
* Fire a weapon such a bow and arrow

**® CORE GAMEPLAY ®**

**Main Game View**

We view the windows console only. The console has code added to keep it at a perfect size to display all content required to play



Figure : Main Game View - Console

The aim of the game is to survive the **Charbies** ( **#** ) and collect all the plus signs **( + )** which are treasure/supplies. One bow will be placed on each map for collection. Once collected you can shoot **LEFT RIGHT** **UP** and **DOWN** using **A**, **S**, **D** and **W.** All enemy objects, treasure, bow and the player are randomly places at start of each level. The player gets to pick which direction they move. If your player object gets killed the game is over and you have to start all over again.

**Core Game Activities**

The player can move around a square map. The map is represented by 40 x 80 character places. Live animated charbies run wild and you have to avoid them or kill them while collecting the supplies/treasure

Once you have collected all the supplies/treasure you can skip to next level. If you die you have to start all over again

**Game Controls**

The player can choose to go up, left, right or down one place each game loop.

|  |  |
| --- | --- |
| **W** | **FIRE ARROW UP** |
| **D** | **FIRE ARROW RIGHT** |
| **S** | **FIRE ARROW DOWN** |
| **A** | **FIRE ARROW LEFT** |
| **LEFT ARROW** | **MOVE LEFT** |
| **RIGHT ARROW** | **MOVE RIGHT** |
| **DOWN ARROR** | **MOVE DOWN** |
| **UP ARROW** | **MOVE UP** |
| **BACKSPACE** | **VIEW PLAYER POSITION** |

**In Game GUI**

Our GUI provides real time with real dynamic information such as your current coordinates, time elapsed or amount of treasure left to collect

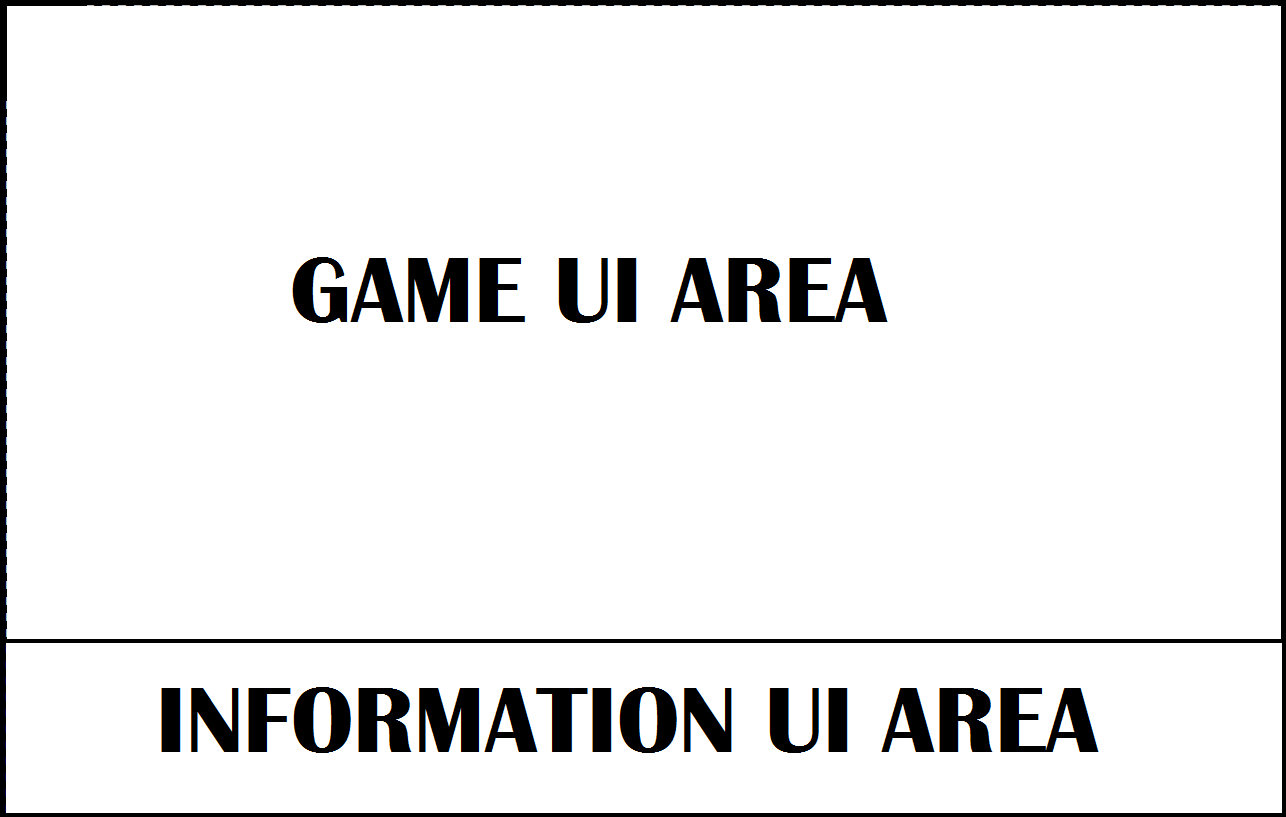


Figure : GUI Style

**® CONTEXTUAL GAMEPLAY ®**

**Game Mechanics**

This game has a number of little mechanics that make some of the functionality possible. Let’s explore it

Object Unique ID generator

This piece of code below in figure 3 gives each object in our list a unique set of coordinates. We use the same code for treasure, bow, enemies and arrows. Line 247 to 253 keeps trying to get a X and Y coordinate until we find it empty. Our random spot is created on our Game Array. Line 254 creates a unique name. Line 255 generates the new Enemy Sub Object. We want a random stamina from 0 to 4. This means these enemy objects get a slightly different access to their update function. Line 257 spawns the object. Now we reset the Serial ID, push object onto our list and place the correct object character on our Game Array from line 260 - 264

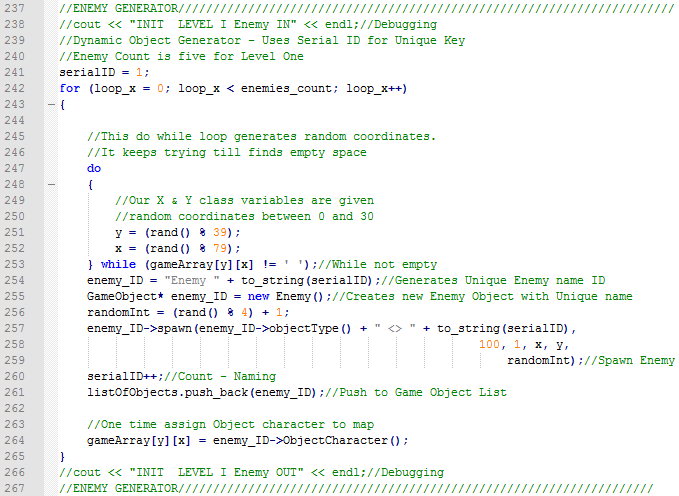


Figure : Unique coordinate algorithm

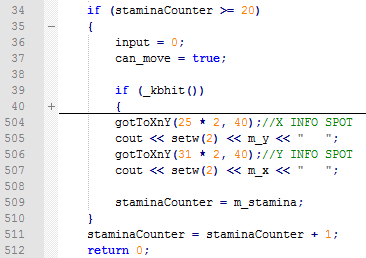
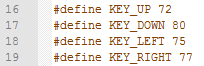
Our user can trigger keys to activate an action anytime. We can limit the user to only have access to a set amount of updates per second. We can set any object to having a specified access to updates per second by setting their stamina. Win figure 4 below we can view our outer rim of our player update function. We say that an object can enter the update functionality when their stamina is 20 or above. If they do not have this value, we skip them past it while incrementing their stamina by 1. In our Main Function we have **Sleep (50).** This is a **windows.h** function and makes the system sleep for 1/20 of a second. This means we have 20 frames per second roughly, depending on sound been called and other various things that might eat away a couple milliseconds during execution of one single game loop. After an object has gained access to their update function, they then have their stamina counter set back to whatever their stamina is. This means if the object has a stamina of 20, then they can access their update function 20 times per second. If they have a stamina of 1, then they can access their update function 1 time per second. We have functionality during level 2 and three where as your collect treasure, the enemy stamina is increased slightly. This creates faster enemies and makes it harder avoid them.

Figure 4 : Key Codes Defined in Game Object Abstract Class

Figure 5 : Get specific character from user input with time out. User must press something or game continues. Windows Function

As for inside the actually function for our player, we cover all possibilities. It’s way too large to share here so let, me share a single portion. We have defined our arrow keys in our Game Object head as each key has a code. Let’s examine the **UP KEY** been pressed

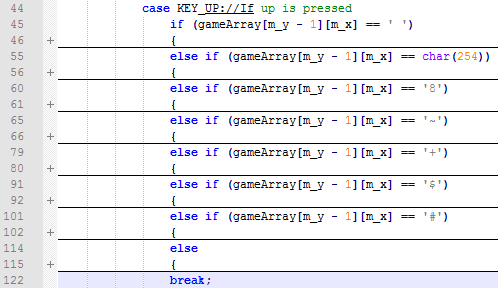


Figure : Player switch direction mechanism with boundary handling

**Map Generator**

Reading characters from a file was drastically faster and more practical that manually adding 3200 characters. Even if we adding an empty space for all the array and then changed the ones that are needed, it would still amount to a huge number. If I was to be honest, this idea was one totally of my own, without any influence. It came from an absolute necessity to make the job go faster. Even adding 3200 characters to a file isn’t easy create the right shape. So I discovered the best thing to do was to create a Photoshop plane that is sliced 50 pixels by 50 pixels. For us this is a character array and looks like this in figure 7 below

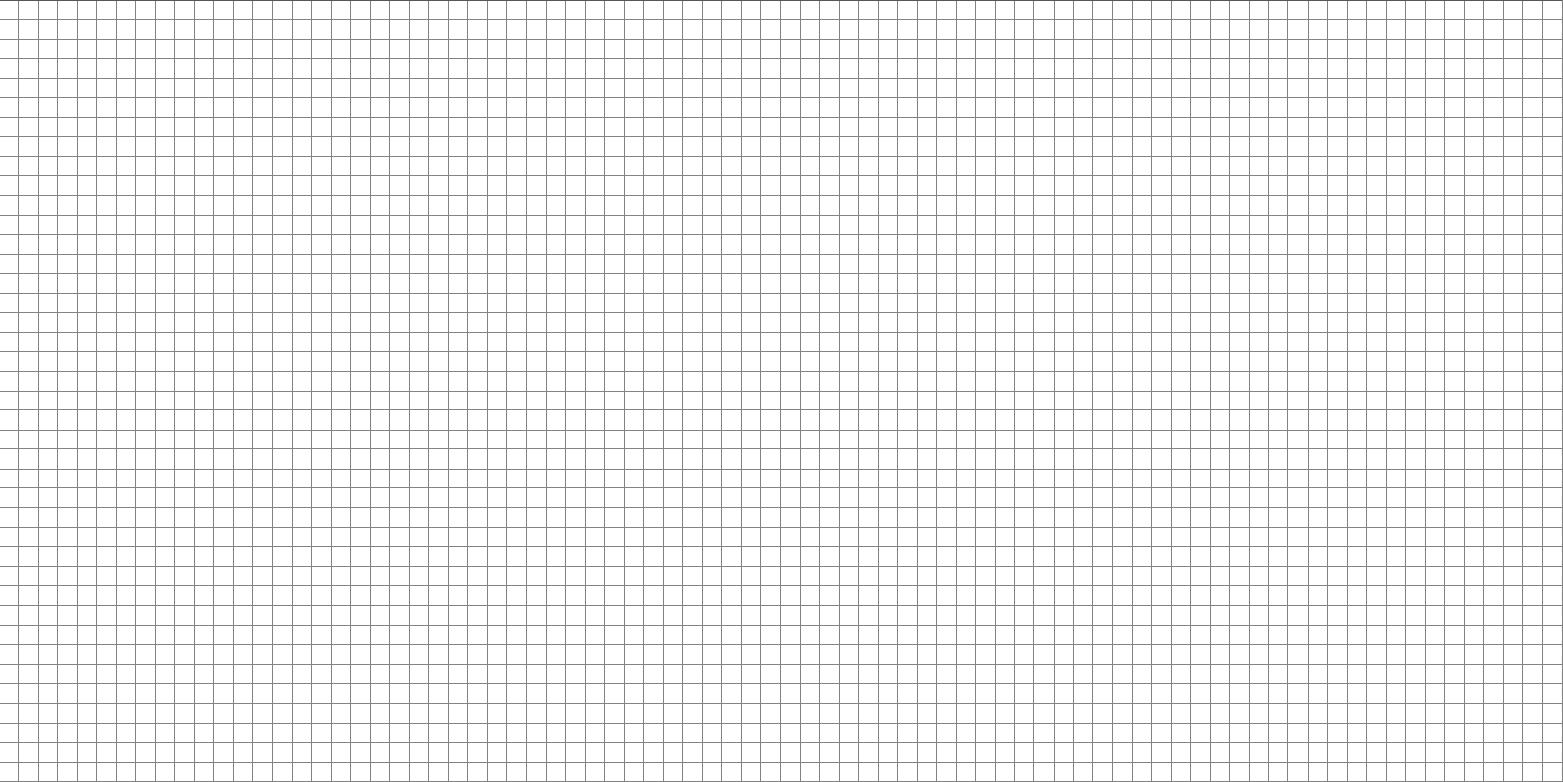


Figure : Photoshop grid layout for creating and designing maps – Part I

Now this grid is virtual. We can then create a text over lay that’s character size fits into this grid perfectly. With this we can create a perfect square pencil at 50 pixels by 50 pixels and start designing out map. Look at figure 8 below. This structure is relatively easy design and draw

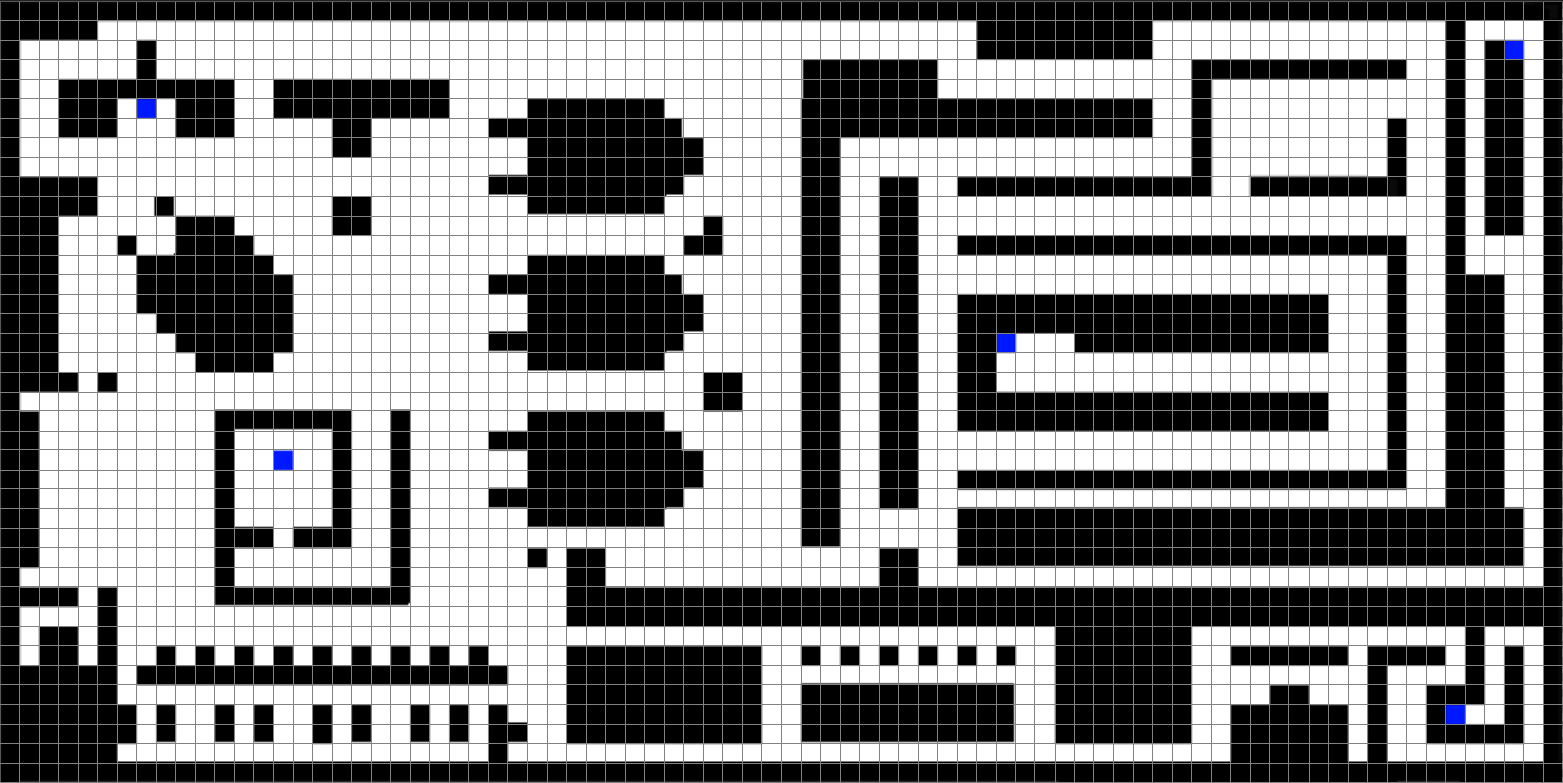


Figure : Photoshop grid layout for creating and designing dynamic maps – Part II

Lastly then we start filling in our character line by line while rapidly viewing the image and rapidly hitting only a couple or few keys.

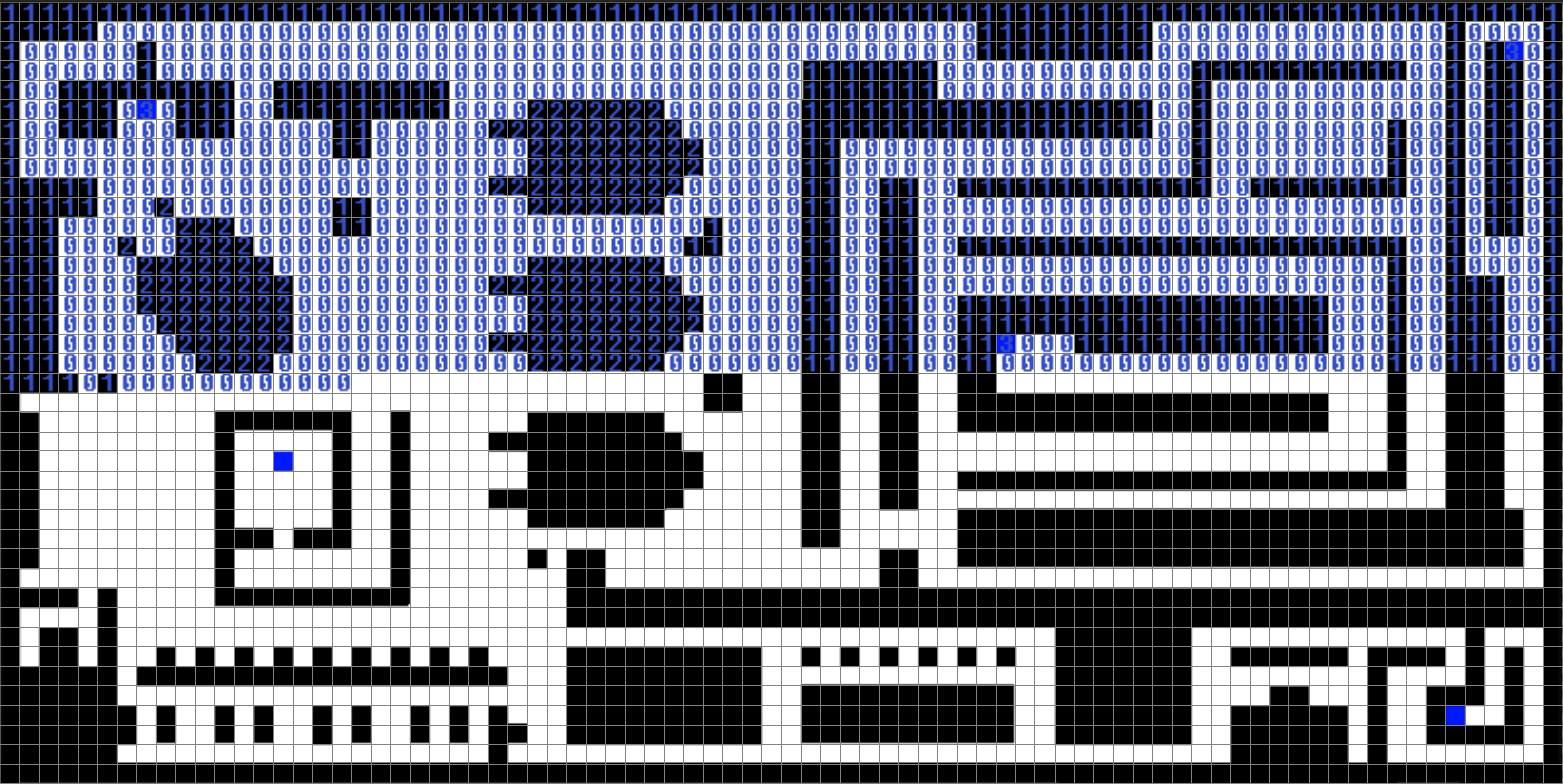


Figure : Photoshop grid layout for creating and designing dynamic maps – Part III

In essence we are looking for the digits only, in the perfect order. Once we are done we copy and paste the digits onto a text file like below in figure 10.

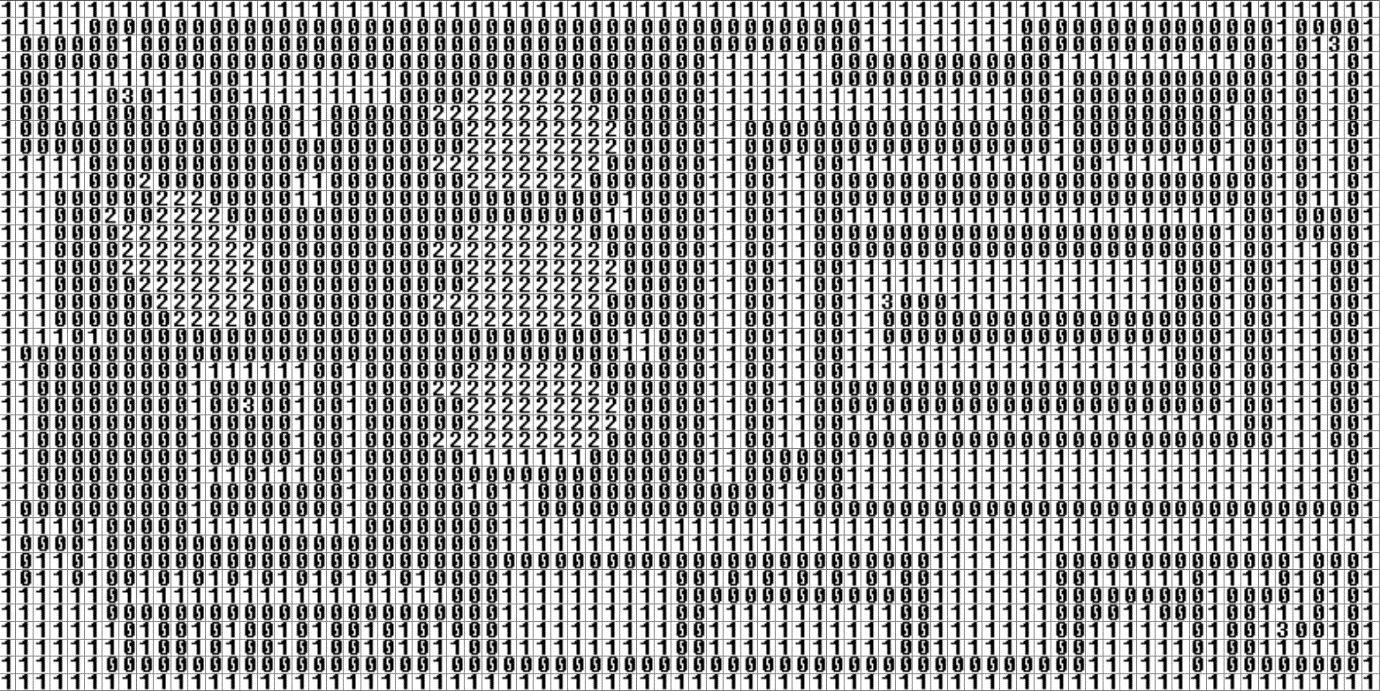


Figure : Photoshop grid layout for creating and designing dynamic maps – Part IV

Lastly we pull these digits into our code via our map generator below in figure 11 from line 198 to 215. We just take in a character at a time. The **fileInput** doesn’t care about return carriages, it read as an entire string. We know its 80 character on X Axis wide so we so go from 0 to 79 (which is 80), then skip line with outer nested loop. Our Y run have 40 lines so that means 0 to 39. Line 198 we read in. Line 203 we condition check the character and then decide what we will add to our game array from line 204 to 212. Note line 212 allows us to add anything such as simple text details for our CUI Information section shared in f8igure 2 earlier. This mechanic is used for each level **include printing our game over and well done message in there functions**

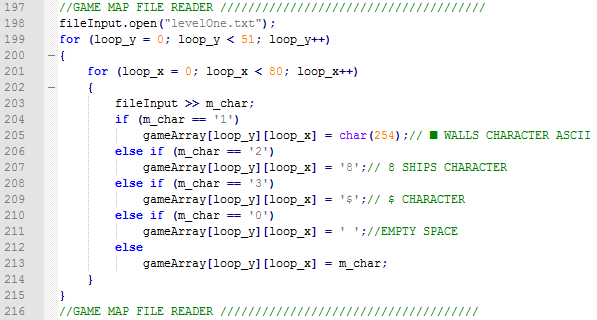


Figure : Photoshop grid layout for creating and designing dynamic maps – Part V

**Path Finding algorithm**

Our path finding algorithm is a complex piece of functionality. We will explain it to a certain extent here. For best understanding it will have to be opened and reviewed.

Our update function for our enemy object is passed down from the GameObject Abstract class. We get passed down a pointer to our main game array which gives us access to check and modify that array from the sub class. We also get the player X and Y coordinates and the level.

This means we can create an algorithm for finding a set of coordinates to the player object during each access an enemy object has to its update function. This functionality is used in level 2 and level 3, but not level 1.

*However, it’s a dynamic behaviour. We will be creating more as this project is only a starter or platform to build a high quality portfolio example of my C++ Object Oriented Skills. The Game will be made available to the public through my GIT Account*

Once we have a few more AI type behaviours we can create very dynamic levels where collecting items or fighting bosses will trigger a change in behaviour for enemy objects. One example is when we are reaching end of level, all enemy objects suddenly become path finder and track you down.

We are an enemy object and we are situated on a X and Y coordinate. We know our coordinates and we know the player coordinates. We have access the game array that tells us everything we need to know about the map. Before we enter the path finder algorithm we save our enemy coordinates onto some placeholder int variables in case anything goes wrong. As an enemy we want to create a path of coordinates to the player object during our single execution of our update function. This means we use an another Coordinates Class that simple has just X and Y Values. This class is specifically used for our path finder algorithm. Our enemy object contains a list and an iterator. We create the perfect path to the player by stepping our enemy object toward the player coordinates. We create a Coordinates object on each step and add it to our Enemy::Coordinates List. Once our enemy coordinates match our player coordinates, which we check at ever path finder loop iteration, we exited the path finder algorithm. Finally, we look up the first coordinates on our list if coordinates and move our enemy onto that position. We have now successfully taking the best step towards the player. We can clear the list of coordinates and exit the update function.

Before we create a Coordinate Object and stepping our enemy onto it briefly, we need to decide what direction we should try. We get the slope of the line between enemy and player, then decide while gathering two most prominent directions by simple comparing X and Y Axis values. Through this we create a combination of directions to attempt to step onto. So we are trying the best steps every time. If the best direction, which is most directly towards the player isn’t available, we try the next one, and the one after and lastly the worse direction which is away from the player the most. Later in Pseudo Code section, we will have a look at some the basic structure we designed the algorithm with

The complexity of a path finder algorithm can increase. We have created a rather simple one and took map complexity and space into consideration while implementing it. This algorithm works, but to create best one that works in all situations. The next step will be to create very tight spots and have the algorithm so that it works in coordination with other objects to firstly as a whole all work out of a grid lock, and secondly to work together to trap the enemy as a group of wolfs. This is the next milestone for the project

**GAME ELEMENTS**

**Characters**

|  |  |
| --- | --- |
| ~ | Bow |
| 0 | Player |
| # | Enemy |
| ■ | Wall |
| > | Arrow **RIGHT** |
| < | Arrow **LEFT** |
| v | Arrow **DOWN** |
| ^ | Arrow **UP** |

**Level / Mission / Area Designs**

User plays as a O who is trying to protect his character camp from the outbreak of the Charmbies who have devourer many of his people already. His mission to collect all treasure(Supplies) and bring them back to his camp. First the O has to find all treasures and fight his way through the map while avoid getting eaten. O has to face three levels of Charbies to complete his mission and return to his family of O’s

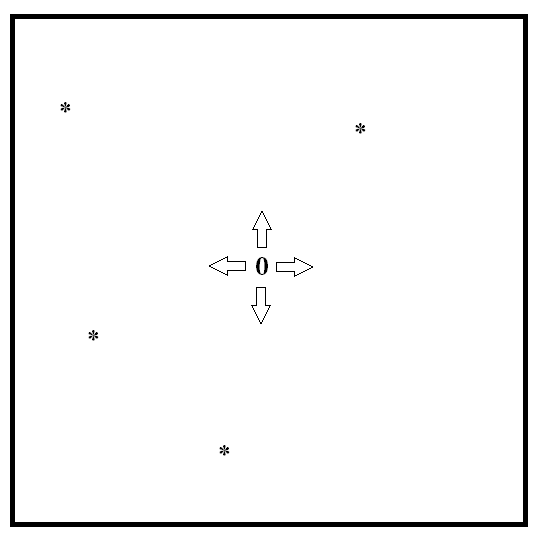
****

Figure : Level Directions Controls

**Protagonist – Player Objects**

The only objects in this game are Player, Enemy and wall

|  |  |
| --- | --- |
| 0 | Player |

**How to play**

The player should try get to all the special treasure/supply characters ‘+’. The player can use the Bow to fight enemies, but has to locate the bow and get it first. Once the player has gotten all the items on first level, they will progress to level two and then level three. This game provides three levels

**Sound Effect**

I have added a handful of sound beeps, with different frequency and length.

**Story**

The O dynasty has become nearly extinct since the appearance of the Hash key Charbies. Your mission is to collect supplies or treasure to bring back to your camp to take care of them. Battle your way through three levels while collecting supplies to bring home. You cannot return home without the supplies

**OBJECT ORIENTED TECHNIQUES**

Our game will have a square array of coordinates. We will have objects that take up one space of this grid. Objects can be an **Enemy**, **Bow, Arrow, treasure** or a **Player**. Each object has various variables such as X coordinate, Y coordinate, stamina or health. Most these values are same for both object types. However, the object types have important differences

The one main element that will differ is there update functionality. The enemy with be controlled by AI while the player will be controlled by user input. The user input uses a special windows function called **\_kbhit()** that times out fast. This allows the game to play away even if the player doesn’t interact. The update virtual function is different for **player**, **arrow** and **enemy**. All objects share many same variables so we can create an Abstract class called GameObject and have all other base object inherit from this parent class. We use composition and polymorphism by utilizing virtual functions or overloading parent functions.

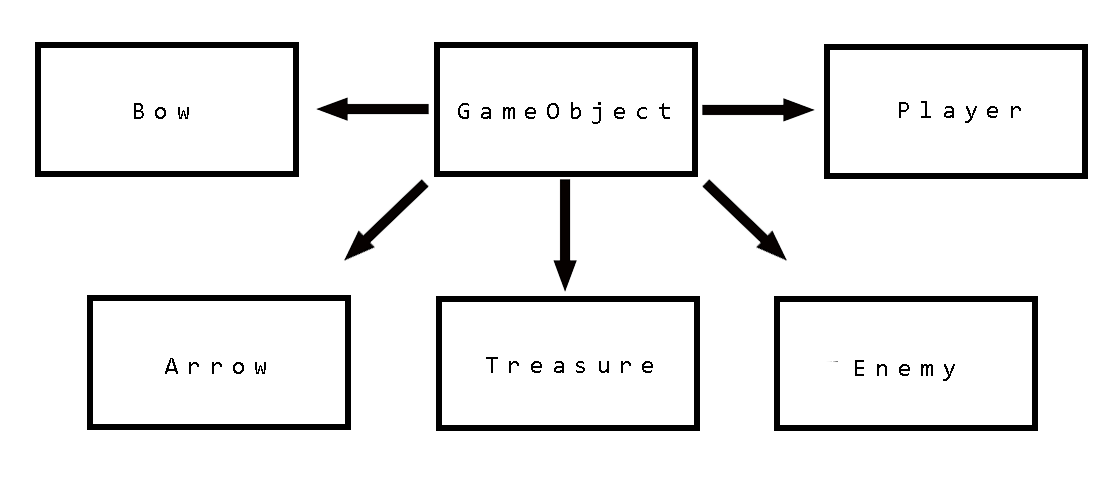


Figure : GameObject & Inherited Sub Objects

When we create a virtual function in our Parent class it allows it be over written by our sub class. Our update class will need to do this. This means if our sub classes have same update class as our parent class we can call the Parent update function and it’s in turn call all the sub classes update function. Each different sub class may have an entirely different update function, and in our case they do. We also pass back through a pointer to our Main game array which comes from the Game Class, player Coordinates and the level we are working on. The function also has a return INT. This can mostly be ignored. However, we do use it for some fucntio0nity. It’s extremely handy to pass back signals from the Child classes, telling the Game Class to commit an action such as destroy arrow, create arrow.

virtual int update(char(&gameArray)[51][80], int player\_x, int player\_y, int level);

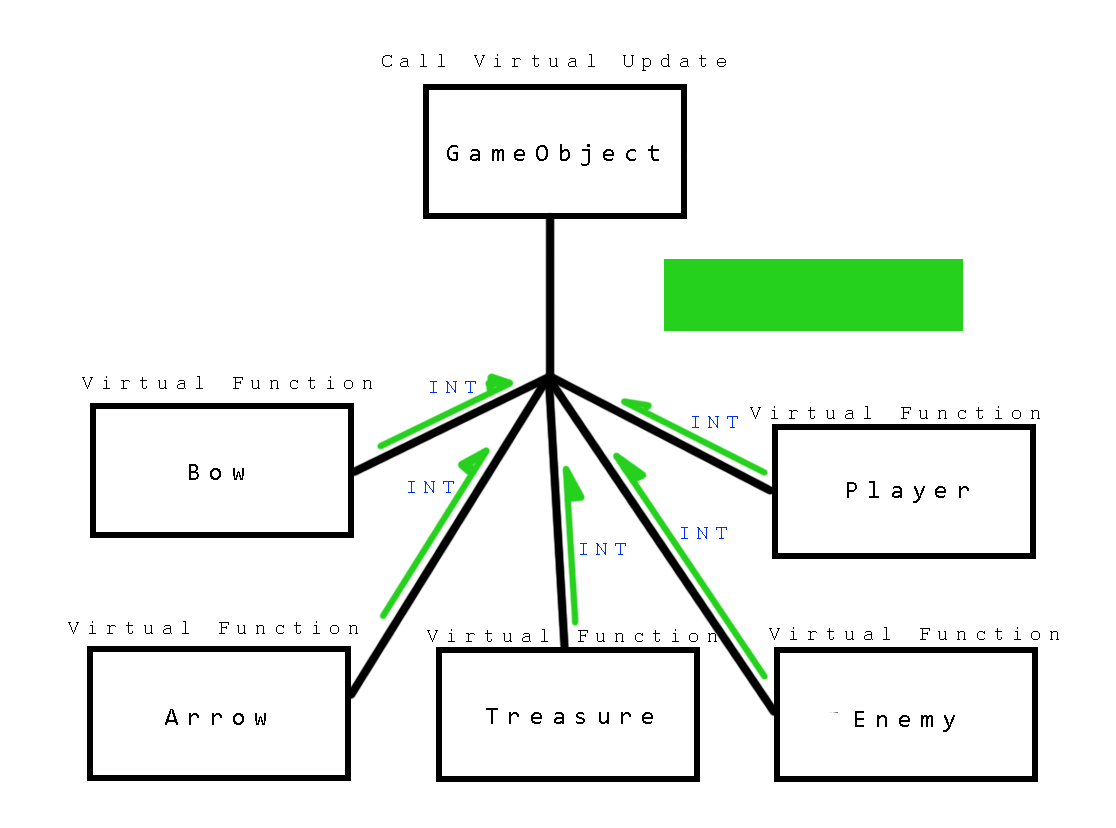


Figure : Virtual update functionality

We use a list of Pointer to objects and iterate through them while calling the update function.

**GameObject**\* P\_Player;

**list** <**GameObject**\*> listOfObjects;  
**list** <**GameObject**\*>::**iterator** iter;

Our Game.cpp Game constructor can now initialize them to a sub class. We do the following

P\_Player = new Player();

We create many more objects in our init class and we can view that description [here](#EnemyGenerator) from earlier in this document.

Once our objects have all be created we can push them onto our list. Arrow objects are created through our return value from our virtual function. When a Player has equipped a Bow and presses either **W**, **D**, **S** or **A** while within the correct distance and scope of other objects, a signal is sent as a return value. Let’s look at our update function in the Game Class

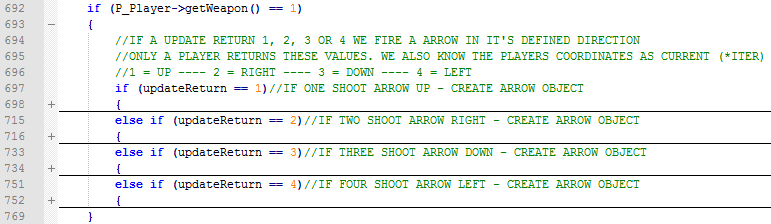


Figure : update return value fires arrow. 1 - 2 - 3 - 4

Let’s have a look into one of these sections and view how the arrow is generated and added to our game list. If our player is armed with a bow and hits any the shoot keys a return value is sent back to the Game Class and an arrow is generated in figure 16 below.

* Line 701 creates the arrow and calls its constructor.
* Line 706 spawns the arrow
* Line 707 assigns its direction which affects its update function
* Line 711 713 places character on game array
* Line 715 adds it to the list and it is treated the same from here onwards as all the other objects
* Line 716 resets the return value for the next iteration

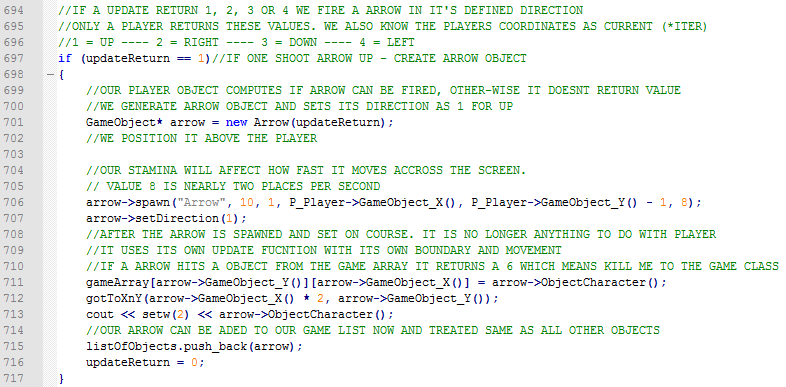


Figure : Arrow generates from return value in our player update call

Lastly we can iterate through the list of objects as if there are same object. This is where our virtual functions come in. Our Game class will have a function called update which with iterate through this list while telling each object to call their own update function. The Player object will skip past GameObject update and onto its own update function. Same with the Enemy object. Our Enemy object update has AI that moves its coordinates while our player has user input to move its coordinates.

Our Game class update function will bypass GameObject update and jump to specific Sub Class function

listOfObjects.push\_back(P\_Player);

**for** (**iter** = listOfObjects.begin(); **iter** != listOfObjects.end(); iter++){  
 (\***iter**)->**update()**;  
}

We can then use this technique for all our functions in our Game class, while utilizing the joys of having all our objects in one list. This is how our very simple game works. Our main simple creates a Game Object. Then through this Game Object, call 5 functions located the Game class. Each function iterates through the list while calling either functions from the parent class of the child class. This is object oriented programming

**As we can see here we are using pointers to objects. Our list is a list of pointer objects. This creates a faster iteration and access of objects**

**PSEUDO CODE**

**MAIN FUNCTION**

Include Libraries and all header files from all other classes

Creates specified size console

Main runs through 3 levels. We call a level via Game Class, complete it and print well done. If the user fail, we print Game Over and exit program

Our main function calls

Game levelOne(1) – creates our game object and ass

levelOne.init() – Call the initialization call which creates map and all objects

levelOne.DisplayPlayer() – Shows the user where player is located

WIN LOSE Functions are called afterwards

**This is done three times, one for each level**

***Our game runs on 20 loops per second. Objects will get access to their update function depending on their stamina. If a player has 20 stamina, then they can update 20 times a second. This results are a swift moving around the map. If an object has 1 stamina is can move one per second. Objects and their stamina will change during game play depending on the level and situation. As a result of this mechanism, we can slow down or speed up enemies or players to give more options during developing the game***

**GAME CLASS**

**GAME::GAME(INT LEVEL)**

Each level is called while adding number 1 to three as parameters. This stands for which level. Each lever has a different map and different amount of enemy’s and treasure to collect. Enemies also move faster and act different depending on which level you are in. Our 3 levels have different enemy count treasure count. The INIT function then uses these to dynamically create enemy and treasure objects while giving them unique locations

**INIT ()**

For each level our init function has a special segment it accesses. Once accessed we firstly read in our level from a text file. We have three different levels. Then after this we create a player object and assign him a random unique location. We pull the enemy count for this level and create that many enemies. Then we do the same fort Treasure. We create one bow and assign the info bar values. Lastly we print it all out one time. Throughout the game play we do not print out the enter array again. We simple move to whatever cursor position we need to modify and simple modify that character alone. This means there is no flickering and game appears to run smoothly

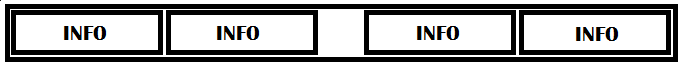


Figure : Information Bar at bottom of GUI - DESIGN IDEA

**UPDATE ()**

Our game update function utilises our Game Object Virtual function. It iterates through list of objects while calling their virtual update. Our virtual update has a int return which is been used for various efficient methods of destroying objects or helping with game flow. As example our 1 to 6 is been used for our arrows. If a 5 is returns we know this arrow has to be destroyed. Another example we have set up returns 1, 2, 3 or 4 as create an arrow objects either of four ways depending on this returned int. 1 is UP, 2 is Right, 3 is Down while 4 is LEFT. We create the arrow object and it then becomes an object in the world that has its own rules. Such as go one place each time it has access to an update. The amount of time an object can access the update function depends on their stamina. An arrow has around 10 stamina, a player around 20 and an enemy varies from 1 to 15

**COLLIDE ()**

We check each game loop for collisions of any objects that therefore need processing. I have noticed by using the Debugger that avoid iterating through list when possible is faster and saves processing time. We have developed ways at declaring an object dead by simple using the game Array rather that checking every object again every object excessive. However, when we still need to we do it here. Player and treasure, enemy and player. As example lets explain how we save using a list iterator.

We pass a pointer to our Game Array back through the classes to the subclasses. This means they can just check array position as they move while using the return INT from function to pass back information or signals of what has happened to the Game class, such as arrow has hit something to destroy it. This is a return 5 value.

We also use return 1,2,3 or 4 for spawning an arrow when user fires one

**CLEAN ()**

We check if an object is alive by iterating through all objects in our list and checking their health. If it is zero or below they are eradicated from the list

**CLEAR SCREEN ()**

This simple clear the buffer when called

**PLAYER POSITION ()**

We have assigned the **BACKSPACE KEY** as triggering this function. Once called it highlights where the player is on the map. This was vital for this game style and nature. It can be hard knowing where you are sometimes. Once a user calls this function it become very clear instantly where you are located.

This is achieved by using a nested loop. We create arrows pointing all along the X and Y axis that the player is situated on. If a player is on X 32, then the entire axis X 32 is filled with arrows’ all pointing towards the player like so. Seems we have the player x and Y coordinates we can easily light up all the right coordinates

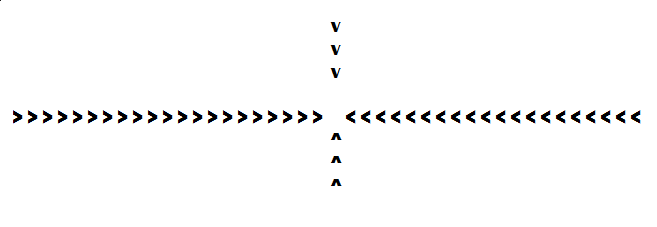


Figure : Player position function design

**WELL DONE ()**

Once a round has been completed, main calls this message to be displayed. The message is read from a file as the design is easier create on a text file

**GAME OVER ()**

Once a game has been lost, main calls this message to be display. The message is read from a file as the design is easier create on a text file

Same Goes for this function as WELL DONE above

**GO TO Y AND X ()**

We do not want to have to clear entire screen every frame. Instead if we can simple jump cursor to a certain character position that we need to modify and then modify it and its clean, faster and more control of game flow as these change can be done from any class or sub class. The buffer display is global access to change. Let’s look at example code below where we are moving the player character one place left after the left key has been pressed

gameArray[m\_y][m\_x] = ‘ ‘;//Make current spot empty in array

goToXandY(m\_x, m\_y)//Just to desired positon

cout << ‘ ‘;//Prints empty space in old spot

m\_x -- ;//Decrement X value by one - This is the objects X value

gameArrayp[m\_y][m\_x] = ‘O‘;//Make current spot Object Character in array

goToXandY(m\_x, m\_y)//Just to desired positon

cout << ‘O’;//Prints out known object character

**GAMEOBJECT CLASS**

Abstract game class has X and Y coordinates, speed, health, ID, type of object and stamina and various other variables.

**SPAWN** function sets the objects x, y, speed, health, stamina and ID.

**GAMEOBJECT\_X** returns x coordinates.

**GAMEOBJECT\_Y** returns y coordinates.

**SET GAMEOBJECT\_X** set x coordinates.

**SET GAMEOBJECT\_Y** sets y coordinates.

**GET HEALTH** returns the objects health.

**VIRTUAL GIVE DAMAGE** is a virtual function which calls sub objects give damage.

**VIRTUAL OBJECT CHARACTER** gets the objects special sub character

**VIRTUAL INT UPDATE** is past down to all subclasses. Along with that we pass down a pointer to our game array, player coordinates and current level. This allows all sub classes direct access to our main game array so that they can compute their movement, especially our AI for Enemies. ALSO NOTE, our update function has now a return value that can be used for various reasons. One example is we can kill arrows by returning 5 when it collides with an object. This saves iteration time in the CLEAN Function. We also can use it for spawning an arrow by allowing the player to use the special fire arrow keys and return 1, 2, 3 or 4 for which direction we spawn an arrow. It’s a useful way and sending signals back to the GAME CLASS

**SET HEALTH** sets the objects health.

**TYPE ID** returns the type id.

**VOID** **GO TO Y AND X** moves console buffer character to a specified row and column

**SET SPEED** sets the objects speed.

**VIRTUAL OBJECTTYPE** is a virtual function which calls sub objects object type.

**INFO** returns all information of object.

**IS ALIVE** returns object dead or alive status

**VOID** **RAISE** **STAMINA** raises stamina by 5

**INT** **GET** **STAMINA** retrieves objects current stamina

**STRING RETURN WEAPON** String returns the objects weapon such as player bow – Intended to have other weapons in game

**VIRTUAL** **ARM** **WEAPON** is used for handling weapons

**VIRTUAL** **INT** **GET WEAPON** is used for handling weapons

**ENEMY CLASS INHERITS GAME OBJECT**

Inherits all functions and variables from GAME OBJECT.

**VIRTUAL UPDATE** function has two types of AI movement to access. It is decided by what level ID is passed down to it. It also gets access to our main game array. One is a pathfinder algorithm and the other is basic random movement. Level one provides the basic random movement while level 2 and 3 provide the special path finder algorithm

**GIVE DAMAGE** Returns a random damage value from 0 to 50.

**OBJECT TYPE** returns the enemy object type “enemy”.

**PLAYER CLASS INHERITES GAME OBJECT**

Inherits all functions and variables from game object.

**UPDATE** function uses a special windows function called **\_KBHIT().** This function allows us have a timeout without player update input. This means the game flows onwards without stopping and waiting for player input. When the player does press a key it is used next time around. The plyer also has a set of keys to use. The arrow keys are for moving while **W**, **D**, **S** and **A** are for firing your arrow if you have collected it. If you fire an arrow it returns a int for the **GAME** **CLASS** to create an arrow object at the correct side of player and set it up. We have a INFO function for our user in case they cannot find where they are on screen. This is used by pressed **BACKSPACE** **KEY**, which sends a **INT** value 6 back to **GAME CLASS** to request it be called

**GIVE DAMAGE** Returns a random damage value from 0 to 100. Generally unused in this prototype so far. I have other currently un used function that are specifically left functional for further development and new game features

**OBJECT TYPE** returns the enemy object type “player”.

**ARROW CLASS INHERITES GAME OBJECT**

Inherits all functions and variables from game object.

**UPDATE** provides the game array so that the arrow may navigate the map while been fired in straight line. Once it hits an object it returns a 5 to signal the game class to destroy it

**OBJECT TYPE** returns the enemy object type “arrow”.

**BOW CLASS INHERITES GAME OBJECT**

Inherits all functions and variables from game object.

For now, the bow class is very limited. We simple just use the object as a placeholder. However, seems this game is being made for further development we are always creating things with further functionality in mind. The bow has many reasons why it should be an object such as having coordinates so we can have it invisible until player gets within a certain distance of it, or even simple in line of sight using the slope of line between bow and player

**TREASURE CLASS INHERITES GAME OBJECT**

Inherits all functions and variables from game object.

**UPDATE** provides the game array so that the arrow may navigate the map while been fired in straight line. Once it hits an object it returns a 5 to signal the game class to destroy it

**OBJECT TYPE** returns the enemy object type “treasure”. Our treasure in this game is medical supplies. We keep the common name treasure for now. We will change it soon to supplies when we have more functionality in the game. Treasure can also be an object for same reasons bow is. One idea is for having treasure invisible until within certain range or within line of sight, just like bow above

UML DIGRAM

This UML diagram is troo large be viewed here correctly. It has to be viewed from the folder called UML found in Assignment Folder which is uploaded.

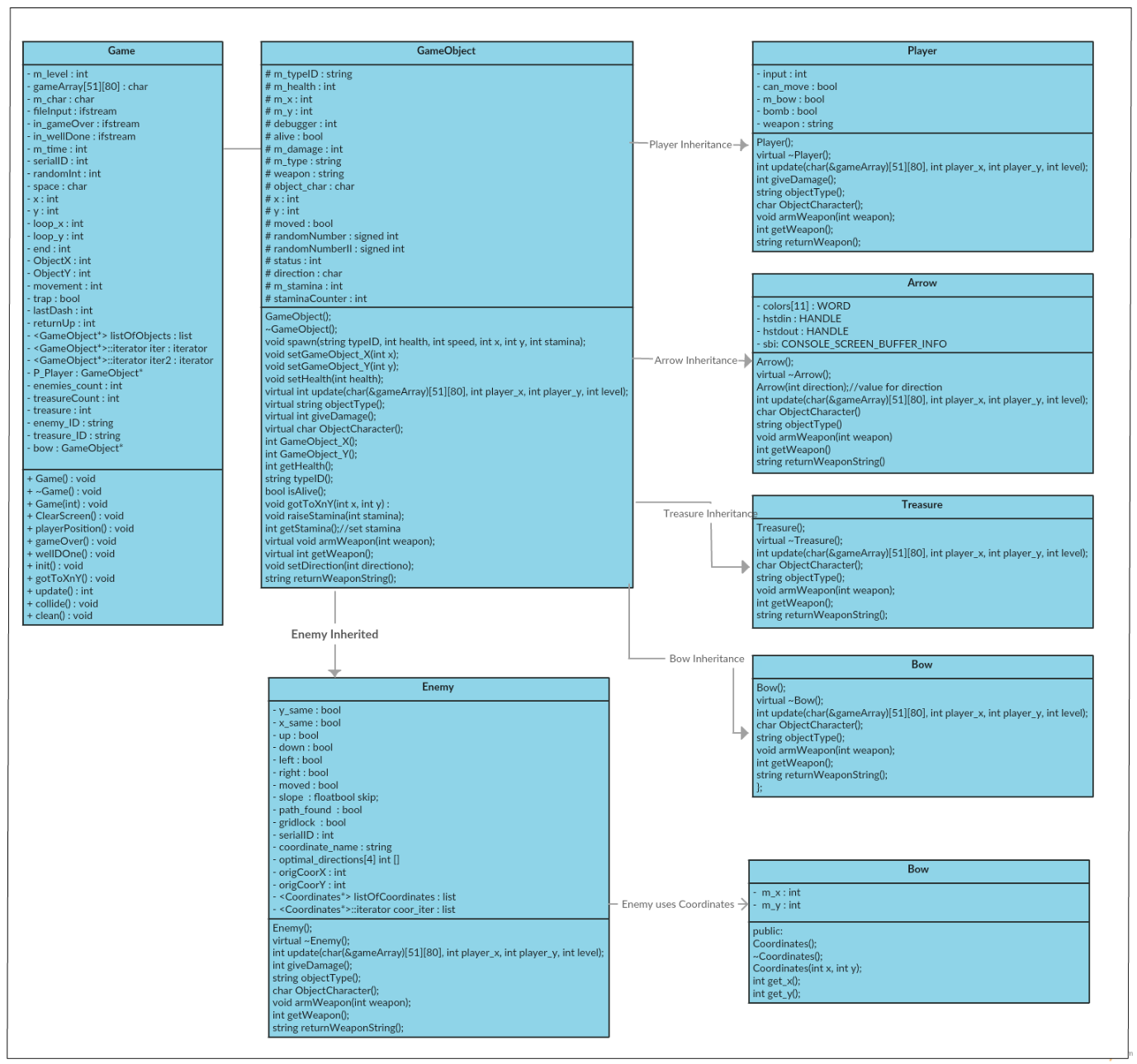


Figure : UML Class Diagram

**GAME FLOW DIAGRAM**

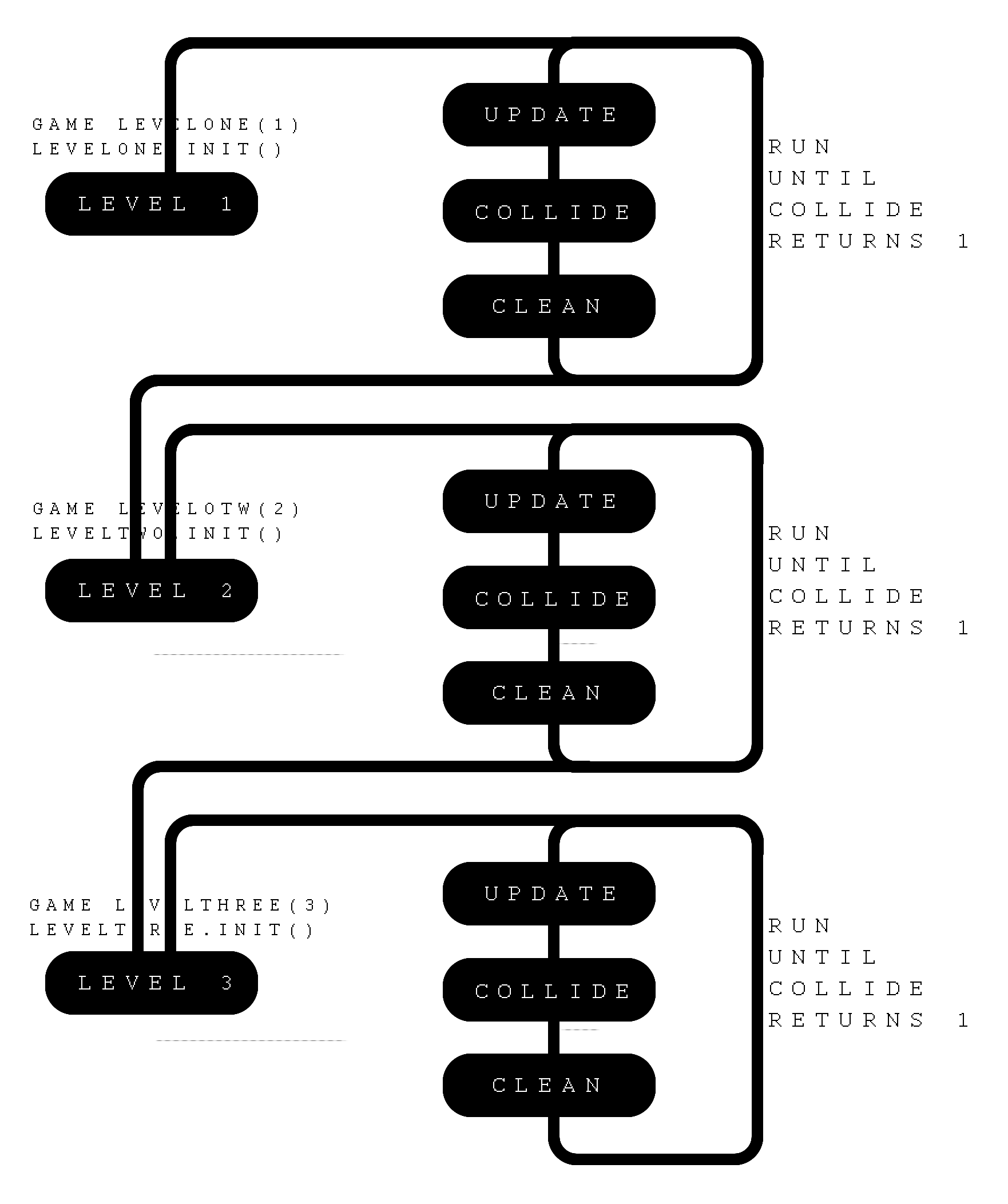


Figure : Game flow diagram

**PSEUDO CODE**

The reason for this section is to demonstrate how I used pseudo code to figure out some of the more complicated algorithms

**Path Finder – Enemy Class – Update Function**

The idea for this path finder is to find a path for the enemy object to the player. Each game loop and each update function will find a new path and move along the best route a step. This means the enemy is always moving towards the player. We do this by passing the level array map, the players coordinate down through the virtual update function. So in the Enemy class, the enemy object has a copy of the actually map and can compute a path and modify their coordinates accurately. This pseudo code below is how I started to try figure out how to do it. I have never tried make a path finder algorithm before so it was fun and very interesting. We use another coordinate class that holds a set of coordinates. We then stick these into a list and compute a good path by adding connected points to the list that lead towards the player and in the end to the player. Once done we move the enemy to the next best step. We do this entire process each game loop, which means each update function. This is created while viewing my map of the first level below in figure 12 and scripting down my ideas alongside it

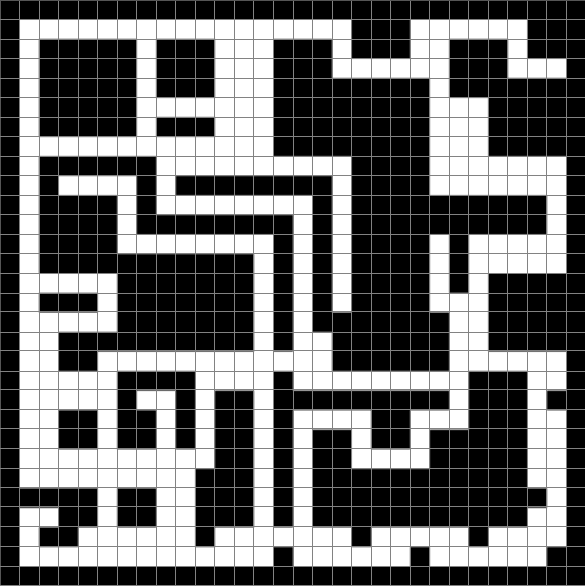


Figure : Level One Design

**PATH FINDER ALGORITHM**

We pass down a pointer to our game array and player coordinates to sub member classes, including our enemy class. This is passed down through the update function

player = (3(X1), 15(Y1))Enemy = (21(X2), 24(Y2))if(e\_Y == p\_Y) Y\_same = true;else if(e\_Y < p\_Y) down = true;else up = true;if(e\_X == p\_X) X\_same = true;else if(e\_X < p\_X) right = true;else left = true;This case left & Up are trueslope = 24-15/21-3 Slope = 9/18Slope = 0.5 We move On X AxisAttempt Left First - then attempt Up - then attempt right and lastly attempt down

We create an array of 4 items. They are directions to try in order. We have four possible directions that we can attempt to move, which means we have 8 possible combinations. Either way we will try to most important directions first and then in defender order of most important. If an object is above us and slightly to the left, then we should first try up, then try left, then right and lastly down

One we have decided upon a combination to try; we then try them in order. The picture below in figure 13 is small size and not intended to study in detail, but we simple need to only see that’s we have 8 combinations. Each single section then breaks up into our 4 ways to try in corrects order of most relevant. This means that our enemy objects are always moving towards the player, by picking best direction to move giving it’s coordinates and objects in its way

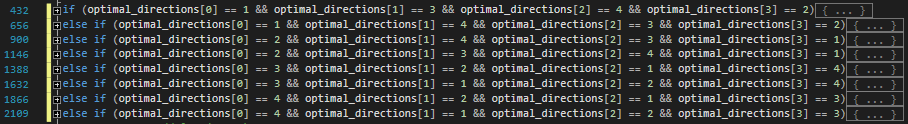


Figure : Pathfinder 8 direction attempt combinations

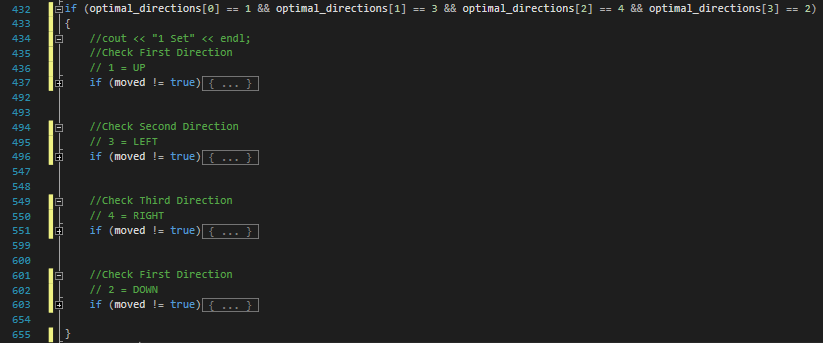


Figure : Single combination attempt

Once our program has arranged best combination we move to that block and we try the most important ones first. In figure. On line 437 we can wee we try up first. If at any stage in there we discover out position up is taken, we simple skip to next one, and on line 495 we try left. We do this until we find a spot

As part of an error handler we only allow the enemy object path finder to create 100 coordinates and then give up and reset. This error handling works well and stops problems even when congested. The enemy objects keep flowing towards the player

If up is true and left is true and the slop is greater than 1. This tells us that we must try to move up first. If that fails, then we try left. If that fails, then we try right and lastly try down. If all fail, we rest and give up. Update will attempt again next round

After we have a combination. We enter one of our sections from figure 13 above. Once we enter it we try our directions in chronical order starting with most important.

We first test if walls, enemies, bows or treasures are in our way. If they are we exit signalling to try next direction in our chronical order, otherwise if they are nothing there, we still need to make sure that also none of our coordinates that we have already created are there. First we check if we have anything in our list of coordinates, if not the we can move to our proposed position. If there are coordinates, we loop through our list of coordinates and make sure none of them are located at our proposed position. Once satisfied, we move there and switch our moved Boolean to true. Before we start this process, we save the Enemies original coordinates in two variables. We move e our actually enemy object along these coordinates as we create the path. If we fail, we exit and set enemy back to original position and empty coordinate list. If we through this iteration check and find that our enemy object is placed on player object, then we have created our path. We switch the path found Boolean to true, and exit the loop. Then we set our enemy coordinates to the first path set of coordinates, which is only on step from where he was. Then we clear the list of coordinate objects and exit the Path Finder loop. We have now moved a perfect step

This algorithm runs for every enemy each time they have access to an update execution

BASIC AI MOVEMENT

We have two types of AI movement. The first was explained above, our Path Finding Algorithm. This one is much simpler. We simple pick random number from 1 to 4. Each number represents a direction of LEFT, RIGHT, UP or DOWN.

**TESTING**

The testing and debugging for this project has to be the largest and most extensive tests and bug searching I have ever done. My first couple of nights, 30 hours+ were spent placing print lines everywhere. I discovered how to use the debugging features for the visual studio and my god, it was amazing. The debugging feature is very useful and speeds up finding the issues. Not only does it speed up finding issues, it displays ones we may never have noticed such as unused variables or some logic that was happening that wasn’t made visible for a normal eye.

Testing involved playing the game consistently after every update. There was a cycle of development for me.

* Make it
* Fix it
* Break it

This cycle went throughout the entire process. Anything that was made was broken, once fixed it then had to be b broken again and again until all possibilities were satisfied. One technique I was using was printing massive amount of data to a file while reading over the file and following objects around the map as they altered their position and went through their update function. This method quickly was swapped for using the debugger. Although I was aware there was a debugger, it sometimes hard moves away from what we know works such as printing lines everywhere. However, after learning to use it and spending literally 40+ hours using it in detail, it because very clear of the enormous usefulness it provides and how accurate and detailed it is. Not only can we use it for debugging, we can use it to understand how long processes are taking. Such as iterating through lists. This functionality has been completely rethink my entire design on many occasions

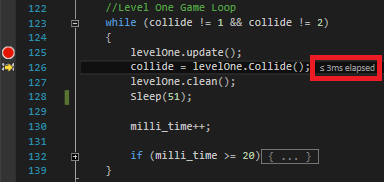
I was at first using a wall objects and passing down a reference to my game list and game iterator so the sub classes could access it. During following execution time, I discovered it was pretty slow and very unnecessary in comparison to other execution of functions. If we look at figure 24 on right, we can see that the collide function took 3 milliseconds to complete. It was a lot larger before I re-calculated my design options and started swapping lists for just plane characters on an array. I also improved speed by instead of cycling through a list to see if objects are colliding, I passed the Main Game Array down through the virtual functions by reference and if an object update function moved them into a collision with another object, the virtual update function has a return int value. This means objects can send signals back to the update function and objects can be set to health zero there and then. It’s like passing the responsibility of an object killing itself by sending a signal to game class who simple sets health to zero. This then allows us to not bother checking for collisions in the collision function in the game class.

Figure 24 : Milliseconds debugger information

Next testing was for the pathfinder algorithm which required leading enemies all around the map until they broke. A break point was set in any suspicious areas and if a break was called we can check line by line to discover what has happened. While we are doing this we can check all game variables, including object coordinates game array and list of objects and their values.

Testing and debugging is a much more pleasant experience with a debugger such as what is provided in visual studio. This is critical to learn as early as possible as the time it saves is astronomical

I had 2 game testers throughout my testing phase which lasted a few hours. I gathered intel and made positive changes. The main issue was the function call we explained earlier here. It doesn’t work well when player is reduced stamina and if you button bash, the list of all keys you bashed in need to be executed and the player will move to all them before. So you have to be patient and not press keys too much

Lastly one great tool for testing and debugging is the replace tool, which can be used on all scopes including the entire project. This saves a lot of time and helped solve a lot of issues fast

**ANALYSIS**

This project has a very small amount of time so it’s expected that lessons will be learned and new plans to make the project better can be worked on after we hand up our best shot. I can see now afterwards that there is much room for improvement. While debugging, you get reminded that there are a lot of variables that are use used and hold a huge sum value as standard because they have not been initialized and do not require as example a normal int, could use a small int or something less expense on memory

Im very happy with many elements of this project. The rapid creation of complex maps using Photoshop and reading in the digits to create huge complex maps that would otherwise take ages. In addition, we can design them very well as we have a high quality graphical GUI to do so as demonstrated in figure 7, 8, 9 and 10 earlier.

The random movement AI algorithm isn’t really good enough. It appears everyone is jigging around at the same time in the same field of reach. Instead we need to swap this for a something a small bit better, such as enemy objects walking a certain direction or path for 10 steps. Then resetting and following another set of steps. These steps should exhibit a rough direction. This means they might go over to an area, then come back, and so on.

The path finding algorithm is as good as could be with the time allotted. However, during design and especially development and debugging, the possible complexity to make it perfect was higher than originally assumed. This is evident in tight areas or if an object goes down a dead end. I feel strongly our Coordinate Objects will need new variable values along with their X and Y values. Each Point will have an attached type. If a dead end is discovered, the route to that dead end can be marked a red, while other more encouraging routes get marked as green. Also congestion requires more though and extra layers of code to deal with it. For this reason, the maps were created very openly and do not have tight spaces. Luckily enough we do have a fall back plan. If any object creates more than 100 coordinates, it drops its plan, deletes everything and skips its go. I have tested this, and it works fine. This simple means if an object is in a tight stop it stays there until an opening appears or the angle towards the player changes to help him move

I’m very happy with the return int virtual function. The method of returning a INT value back to the game class has huge potential to avoid have to check lists and instead do many things in one go. We simple put do more at every execution where possible while we are there. Also passing down the reference to the game array was very useful. It’s a perfect example of using a pointer and how we can use them to our benefit in a OO environment.

The special window function that we explained earlier [here](#inout), that allows us to use a input command from user while continuing to execute the game has one flaw. If we press the left key 10 times and stop, we still have to wait for all 10 times to execute. This means if you are impatient and keep hitting the key, when you reach the point and want to now go up, you may press up, to find it keeps going left all them times you pressed it. So it stores them all in the buffer and goes through them in the order you pressed them. We need to set it so it only ever holds one key at a time. If we press the key again, it deletes that saved key and replaces it with a fresh key. I will research this and find a solution. This isn’t actually a problem until you slow down the player by reducing their stamina. Then you will find players button bashing. My two game testers

There was problem using the SCII Extended set. I also got conflicting values from different pages and ended up dumping a lot of them and using the old 7-bit set. I did keep the wall as it never caused any issues yet

**CONCLUSION**

This project is by far the most fun I have had programming to date. I can clearly see now why were assigned the project and the benefits of going through it. It’s really about objects and not graphics. It’s to show us the idea of object orientation and give us some first-hand experience developing a larger object oriented application using simple keys. It’s a very elegant touch and was much enjoyed. In addition, it serves as a platform to demonstrate high skill object orientation game design and development without having the works about the graphical side of it. We can demonstrate our programming skills to a very high level with just a ASCII key set and a console

I really enjoyed the project and really look forward to developing my much more advanced ASCII Extended console game. I wish to create a really solid example of great algorithms and a great game for my GIT Hub portfolio for trying to secure of job as a programmer in a game development studio later in the years. It’s the perfect platform to demonstrate algorithm skills and general programming skills as game developer

Thanks for Reading