Contents

2D Zelda-Style Game “Dungeon Dave”

Specification

The overall design goal is to create a game which contains many elements from classic top down platformers, but is suited for the modern mobile market. Most mobile games follow a similar pattern in terms of the how they engage with the users; games with short playtimes, but with a lot of replayability. One of the ways this is achieved is by using procedural generation. This process allows for parts of the game to be created dynamically following a pre-determined set of rules. For example, in the game “Minecraft”, the entire world is procedurally generated block by block, so that the player can have a new environment to explore. Procedural generation could be used to produce the levels of my game so that each time the game is loaded it is a slightly different experience. This would keep players engaged and would be well suited to the current mobile market.

Classic top down platformers such as “The Legend of Zelda” series often have a similar feature set. The player can move in one of the 8 cardinal directions, there are enemies which can hurt the player, the player can fight and destroy the enemies and there are collectable items which can aid the player. The puzzles the player faces are usually minimal and are solved by destroying enemies or collecting certain items.

Considering these tropes; the type of game I decided to design was a procedurally generated dungeon crawler.

The dungeon would be theoretically endless and each floor would consist of a number of procedurally generated rooms. The layout of each room will also be handled procedurally, to vary the internal structure, possible collectables, traps and enemies. Once the player has “beaten” a floor a new floor would be generated with a few more rooms to increase the difficulty. This would increase replayability as the player is presented with a different maze each time they play and it also gives them a goal to try to reach a deeper floor than they previously have. Rather than having images for predefined rooms, this approach would be best suited to using a tile engine. This would allow for a small set of tiles to be used to create a more complex and dynamic landscape.

The player should feel quite underpowered compared to the enemies and this could be achieved by limiting the player’s offensive capabilities. There is a 2D multiplayer game called “Towerfall”, which pits 4 players against each other, but their only weapon is a bow and they each only have a single arrow, which can be fired and recollected. This leads to interesting player behaviour, making them more conservative in their attacks and generally leads to more defensive tactics. Giving the player a single arrow seems too limiting in my case, however giving the player only a single ranged attack method and a very limited amount of reusable ammunition would force the player to be more conservative. Overall this will help with the tone of the game in giving the dungeon an oppressing feel.

As well as being able to hold an amount of arrows, the player should also have an amount of health which can be reduced by being hit by enemies or walking onto traps. Much like the classic Nintendo games, I want to use a heart system to represent the player health with each half of a heart representing 1 player health. Along with the displayed arrow count, this would lead the user interface to have a similar style to that of some of the earlier “The Legend of Zelda” games;

“A Link to the Past” UI

Notice how the UI is kept quite simple, so that the player can quickly tell how many arrows of how much life they have remaining. In terms of the UI, it would also make sense to provide the player with a mini-map. This would show what room they are in the dungeon and also show the layout of any surrounding rooms. This will be particularly essential when using procedural generation as the levels will start to get quite large, and without a map of some type the player will easily get lost and frustrated. A good way to represent a map of the rooms is demonstrated in “The Binding of Isaac”;

“The Binding of Isaac” Mini-map

This small mini-map provides a large amount of information to the player. The darker rooms are the rooms unvisited by the player, the lighter rooms have been visited and the other symbols represent what the room is or what collectables can be found within the room. The player can use this to determine where they should go next, or to find a particular item they are looking for, or even to just back track to a room they have previously discovered. A mini-map is almost essential when dealing with mazes of rooms and is certainly something that I need to include within my game.

The player control options should be kept quite simple in order for the game to be best suited to mobile platforms which have limited amount of input options. The player should be able to walk in the 8 cardinal directions (for example, walking diagonally north-east would require the player to press the “up” and “right” directions together). They should also be able to fire an arrow; however, this should not be a simple button press. When using a bow, it should be drawn first and then released to be fired, so it would make sense for the user input to model this action. When the player holds the “fire” button, the character will begin to draw the bow and then hold it ready to fire. When the player releases the “fire” button the arrow is then shot from the bow. This gives a more realistic feeling to shooting an arrow and helps to increase tension as the player needs to allow time for the bow to be drawn before a shot can be taken.

It would also make sense to only show the current room the player is in on the screen at any given time. Showing other rooms which currently have no effect on the game seems redundant and could even be confusing to the player. Instead as the player leaves one room and moves to the next; the view should transition to centre on the new room the player has entered. This technique would also allow items and enemies within rooms to only be active when the player is present, which would save on computational power and memory usage.

The way in which the player progresses between floors is also important. If the player had to simply find a stairway down to the next level it would present no challenge, and would lead to players rushing to the next floor before spending time to explore the current floor. A solution to this would be to have the exit to the next floor “locked” until the player has found the key(s) required to open it. The amount of keys needed to open the exit to the next floor would depend on how many rooms have been generated on the current floor. This would mean for larger floors the player would have to explore a similar total percentage of rooms in order to find all the keys necessary to open the exit.

It would also remove some of the challenge if the player could just bypass any enemies they don't want to fight. For example, if a player enters a new room with only 1 health left, and there are 3 enemies in that room, the player could simply run to one of the exits or even back the way they came. This could allow the player to explore the dungeon fully whilst avoiding conflict which would make the game too easy and the enemies almost irrelevant. Instead it would make sense that if the player enters a room which contains any enemies; the doors to the room are locked until all the enemies have been defeated. This forces the player to fight, and further increases the value of the arrows the player is holding. For example, if the player was locked in a room with 3 enemies, but only had 1 arrow, they would have to recollect it after every attempted shot, leading to a very difficult fight.

The enemies themselves should have two states in which they function. A passive state where they are simply patrolling the room and an active state when they are pursing the player. The trigger for switching the enemies state could be based on the proximity of the player, giving the player the option to try to avoid direct conflicts where possible and take a stealthier attack approach.

So overall the game will have the player traversing multiple floors of procedurally generated rooms. In order to progress to the next floor, the player will have to find the appropriate amount of keys in order to open the exit. To hamper the player progress there will be traps and enemies which can damage the player. Whilst traps have to be avoided, the player must destroy all enemies in order to leave the current room. The only weapon the player has at their disposal is a bow and a limited number of arrows. The goal is to try to reach the deepest floor possible before the player is eventually killed. A mock-up of how the game could be displayed would look like;

Mock-Up of Game Design

The main section of the screen shows the current room the player is in. As stated earlier, the doors to other rooms are currently all closed as an enemy is still alive in the room. The structure of walls in the centre, will be procedurally generated for each room to stop the rooms feeling too similar. The "Pacman" ghost represents an enemy in the room, and the hero sprite represents the

player character. The enemy would patrol the room until the player was discovered. The heart on the floor next to the enemy is a pick-up which can be used by the player to recover some health. How many of these pick-ups are in each room will also be decided when the rooms are generated. Also note the user interface; the player health is an overlay on the main game screen, whilst the mini-map and arrow count are displayed on the bar to the left. The health simply shows a full heart to represent 2 health, a half full heart to represent 1 health and an empty hearth container to represent 0; so in the case above the player would have 4 health out of a total 6. The arrow count would just show an arrow for each one that the player is carrying. Finally, the mini-map would show 9 rooms with the centre room being the current location of the player. This saves having the mini-map having to try to display the entire map which could be difficult as the floors get larger or just confusing for the player to understand.

Game Design

The best way to document the design of the game is to follow the process through from start to finish. This makes it clear how certain features are built upon and can also show how improvements are being made as the design progresses.

As I was planning to use DirectX (specifically direct draw) to handle the video memory management; before any of the game design began, I first used a template to create a working DirectX environment. This template handled the initialisation of the window, and the necessary DirectX objects. These objects include the direct draw object, and three direct draw surfaces. The main operations of direct draw are handled through a process called surface flipping. Firstly, any images which are used in the program are transferred from their bitmap to one of the direct draw surfaces, in this case *"DD1"*. The program can then reference areas on this surface to create individual sprites and arrange them on another surface. These sprites are then placed onto the back surface, known in this case as *"DDB",* which is just a memory buffer not being printed to the screen. The current frame being drawn to the screen is stored on the primary surface, known in this case as *"DDP"*. Once *"DDP"* has finished drawing to the screen and all the necessary images for the next frame have been stored in *"DDB"*, the surface flip can occur. The surface flip reallocates the memory of the back surface to the primary surface and vice versa. This means that the previous front surface is now known as *"DDB"* and the previous back surface is now known as *"DDP"*. This process can be repeated indefinitely to keep producing new frames, without having to actually move large amounts of memory. This means that a loop within the program can be used to input user controls, calculate any collisions or in game changes, and then draw these changes to the back surface ready before flipping the surfaces. It is within this main render loop that the majority of the game code will be executed.

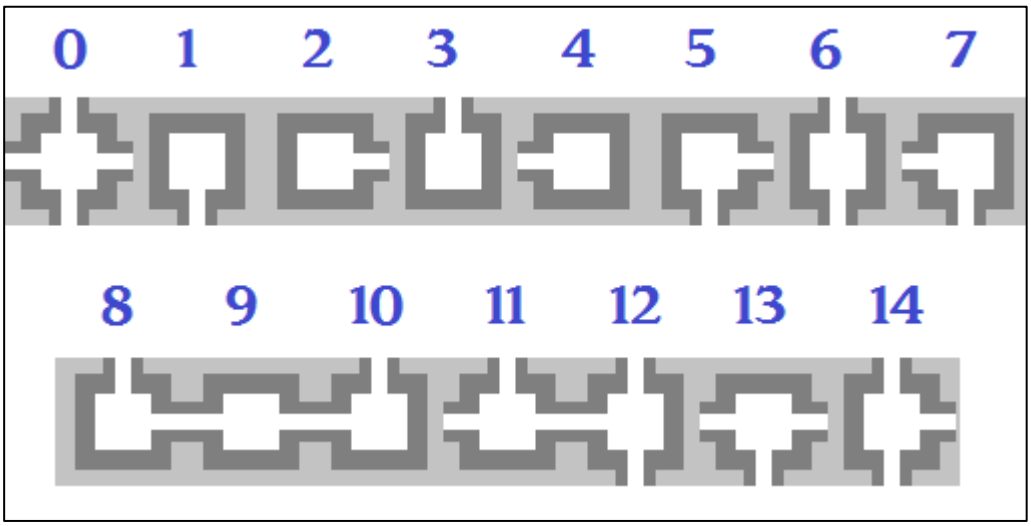
When first starting to design the game I first considered how I would approach creating the world, and how the logic behind my procedural generation would function.

Procedural Generation

The end of goal of the procedural generator would be to produce a single floor of rooms which all connect without any mismatched paths or paths open into empty space. It also needs to be able to create the total amount of rooms specified by an integer number.

The approach I took was to consider rooms which only connected to other rooms in the 4 cardinal directions (N, E, S, W). This would allow me to place a room with a "north" path below a room with a "south" path and they would defiantly join up. This saves on the added complexity of trying to have paths which are also created dynamically. In total this gave 15 different possible

"Room Types" depending on where they had paths leading to (the 16th possibility would be no paths to any rooms, which would be useless).



Representation of my "Room Type" scheme

The theory behind this generation system is;

- Randomly choose one of the above rooms and place it in the centre of an empty floor grid.
- That room and any connecting rooms are then examined to see which other spaces in the surrounding grid has open paths.
- From this list of possible new room positions, one is selected at random.
- A viable Room Type then has to be selected for the new room based on the surrounding positions in the grid.
- That new room can then be stored, and as long as the required number of rooms hasn't been created the process can start again from inspecting the first room.

This process should allow a floor of different room types, as shown above, to be created up to a limit of rooms as specified by the program. I broke this process down into four functions; *"StartGenerator", "CheckSurroundingRooms", "ChooseRoom"* and *"SpawnRoom"*.

To create the grid of rooms; a struct was used to represent a single room and then a two dimensional array of 100 by 100 rooms was declared.

```
struct Room {
    bool occupied;
    bool examined;
    int roomType;
    int roomContents;
    int northPath;
    int eastPath;
    int southPath;
    int westPath;
};

extern Room floorGrid[100][100];
```

Each room has two Boolean values, *"occupied"* is the active state of the room, if it's true the room exists and if it's false it can be considered an empty space. The other Boolean, *"examined"*, is used by the *"CheckSurroundingRoom"* function, so that when it is checking through all the connected rooms it doesn't check the same room twice.

The integer *"roomType"* represents the corresponding image above and lays out the exits/entrances of the rooms. The next variable *"roomContents"* has four possible states; 0 if the room is a default room, 1 for the starting room, 2 for a key room and 3 for the exit room. These values are used to populate certain rooms differently.

The final four integer variables all represent the state of an exit/entrance in the given direction from the room, and each can have three possible states. 0 is when there is wall, 1 is a path which has not yet been connected to another room and 2 is a path which is already connected to another room. These values are used when checking rooms for possible paths and also when decided what viable rooms can be chosen for a given position. Whenever a new room is placed a function is called which can relate the room type to the corresponding paths and sets these 4 variables. This function is called *"SetRoomValues"* and it has three parameters; the room type, and the x and y co-ordinates of the new room within the floor grid. It marks the room as occupied, sets the room type, and appropriately defines the path values. The first section of this function is;

```
void SetRoomValues(int roomType, int roomX, int roomY)
{
    floorGrid[roomX][roomY].occupied = true;
    floorGrid[roomX][roomY].roomType = roomType;

    if (roomType == 1) {
        floorGrid[roomX][roomY].northPath = 0;
        floorGrid[roomX][roomY].eastPath = 0;
        floorGrid[roomX][roomY].southPath = 1;
        floorGrid[roomX][roomY].westPath = 0;
    }
```

Start Generator Function

Now that the floor grid has been defined, the first step in the process is to randomly place one of the room types above in the centre of the grid. The start generator function is an initialiser, and sets up all the necessary procedure to begin the procedural generation. Firstly, as the pseudo random number generator will be used to help define a lot of the features of the dungeon, it is seeded with a time value. A random integer is then chosen between 0 and 14 to represent the room type of the first room, this achieved using the mod operation. As this is the centre room in the grid it needs to be placed in the floor grid at position 50, 50. With this information the room values can be set using the set room function and the room contents can be set to 1 as the first room generated will always be the starting room.

Once this first room is marked as occupied on the floor grid, the check surrounding rooms function can be called to inspect it for open paths. Finally, the choose room function is called to select a room for the new position. This function appears as;

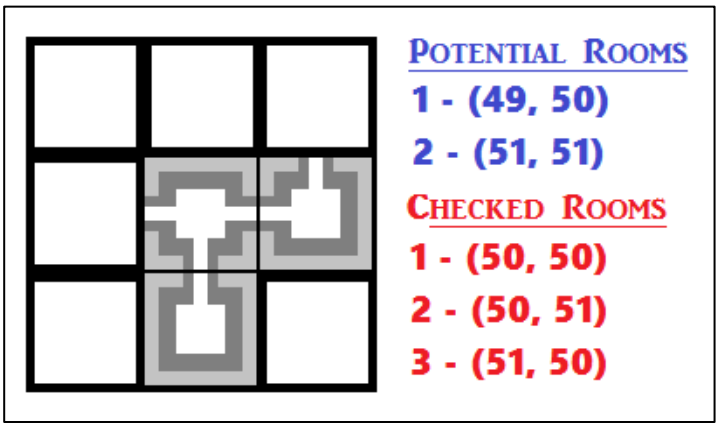
```
void StartGenerator() {

    srand(time(NULL));
    int i = rand() % 15;

    floorGrid[50][50].roomContents = 1;
    SetRoomValues(i, 50, 50);
    roomCount++;

    CheckSurroundingRooms(50, 50);
    ChooseRoom();
}
```

Check Surrounding Rooms Function

This function is passed two parameters; the x and y co-ordinates of a room in the floor grid. This function has two main functions, first it checks the all the directions in the room it was presented and checks for any open paths. If an open path is discovered the co-ordinates of the new potential room are stored in a struct called *"RoomCoord"*, which just holds the x and y co-ordinates of potential new rooms within the floor grid. To cover the large amount of possible paths, an array of 1000 room co-ordinates is declared as *"potentialRoom[1000]"*.

The second part of this function checks all the directions in the current room for paths which are already connected to another room. If another existing room is discovered, then this function is called recursively to check that room as well. To stop this function checking rooms which have already been examined, the room variable *"examined"* is set to true once a room has been investigated.

This means that when this function stops running it will have checked every room which is already placed on the grid for open paths and stored the co-ordinates for any potential new rooms in the potential rooms array. As the next room to be generated is chosen from this list at random; the order which the existing rooms are traversed, and potential rooms are added to the array is irrelevant.



Example "Check Surrounding Rooms" function with discovered positions

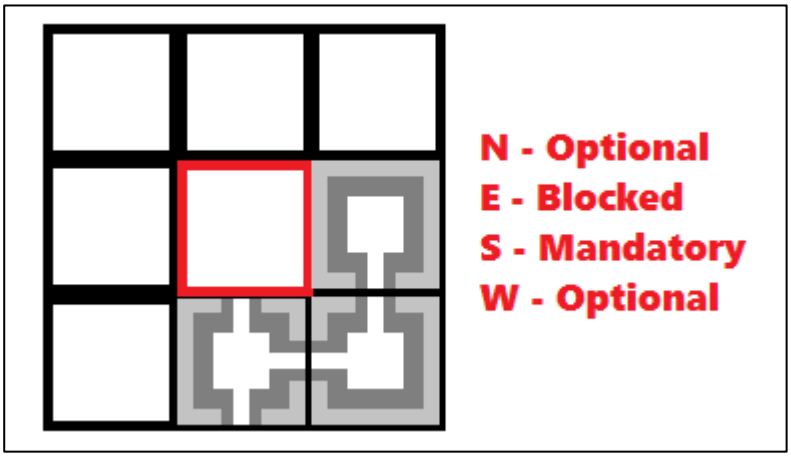
Consider the layout above, assuming that the centre room is the first room located on the floor grid at position (50, 50). If the function was called on this layout it would first check the centre room and discover the 1st potential new room to the west at (49, 50). It would then discover the two connected rooms and recursively call itself again for (50, 51) and then (51, 50). The room to the south (50, 51) is inspected and found to have no open paths and only one connecting room that has already been examined. The room to the east (51, 50) is then checked and the last potential room (51, 51) is discovered and added to the list. A segment of code showing both of the checks performed by this function is;

```
if (floorGrid[roomX][roomY].westPath == 1) {

    potentialRooms++;
    potentialRoom[potentialRooms].roomX = roomX - 1;
    potentialRoom[potentialRooms].roomY = roomY;
}
//Mark room as examined, so the room checker doesn't get caught in a loop checking the same rooms
floorGrid[roomX][roomY].examined = true;

//Recursively call the function to check all connected rooms
if (floorGrid[roomX][roomY].northPath == 2) {
    if (floorGrid[roomX][roomY + 1].examined == false) { CheckSurroundingRooms(roomX, roomY + 1); }
}
```

Choose Room Function

The purpose of this function is to select the co-ordinates for a new room from the potential room list, and then select a viable room to place in that position. First a random position is chosen from the potential rooms array, and the co-ordinates for that room are stored in local variables *"newRoomX"* and *"newRoomY".* The potential rooms array is then cleared ready for the next time it is used to check the surrounding rooms.

The rest of this function examines the position of the new room and chooses which room type(s) could be used. This is quite a complex function due to the large number of possible scenarios available. Each of the 4 cardinal directions around the new room position could be in one of three possible states. There could already be a room in the position with an open connection, this is a mandatory connection which the new room must connect with. There could be a room in the position which has no path to the new room (a wall), which means that direction must not have an open path. The last scenario is that the position is currently unoccupied, which means that having a connection with the new room is optional. Consider this scenario for a potential new room;



Example Scenario when choosing a viable room

In this example the new room type must have a south path, mustn't have an east path, and can have any path in the other two directions. From this data, any room types which don't have a south path or do contain an east path can be considered not viable, this leaves the viable rooms for this space as; 1, 6, 7 and 12.

The function handles this on a per case basis using nested "if" statements which makes this the largest section of code within the generator. Firstly, 8 local Boolean values are set; 4 to represent any required connections and 4 to represent any blocked paths. These values are then used to describe the conditions for all the possible layout scenarios.

The outer conditions check the 15 possible cases for the surrounding positions being blocked or not. The inner conditions then check the unblocked positions for any mandatory connections. Using these two checks each possible scenario can be covered. Once the appropriate scenario has been selected for a position, one of the viable rooms is chosen randomly and the *"spawnRoom"* function is called to add the room to the floor grid.

During early testing of this technique I discovered that sometimes the rooms would stop generating before the designated number of rooms had been reached. This was because if lots of rooms which only had one path were selected early by the random number generator then a scenario where there are no new open paths could occur. To solve this; when the current amount of rooms is less than half the required amount of rooms, only rooms with two or more connections are

selected. This makes sure that the room generator can't run out of paths in the first half of generation as each new room will always guarantee at least one new open path. An exert of the code from this function showing the conditions and the room selection is;

```
//Case 1
if (northBlocked == true && eastBlocked == false && southBlocked == false && westBlocked == false) {

    // Case 1 - East Connection
    if (eastConnection == true && southConnection == false && westConnection == false) {

        if (roomCount < roomMax / 2) {
            i = rand() % 3 + 1;
            if (i == 1) { SpawnRoom(8, newRoomX, newRoomY); }
            if (i == 2) { SpawnRoom(9, newRoomX, newRoomY); }
            if (i == 3) { SpawnRoom(11, newRoomX, newRoomY); }
        }

        if (roomCount >= roomMax / 2 && floorGrid[newRoomX][newRoomY].occupied == false) {

            i = rand() % 4 + 1;
            if (i == 1) { SpawnRoom(2, newRoomX, newRoomY); }
            if (i == 2) { SpawnRoom(8, newRoomX, newRoomY); }
            if (i == 3) { SpawnRoom(9, newRoomX, newRoomY); }
            if (i == 4) { SpawnRoom(11, newRoomX, newRoomY); }
        }
    }
```

Spawn Room Function

This function sets the values for a new room in the floor grid. It also has to set the path values for any newly connected rooms, so that the paths are marked as connected rather than open. Finally, if the current count of rooms has not reached the desired maximum amount of rooms, then the check surrounding rooms function and choose room function are called again to place another room.

There are two global variables which are used by the spawn room function; *"roomCount"* and *"roomMax"*. Every time a new room is added to the grid by this function the room count is incremented by one to represent the current total of spawned rooms. The room max variable is a predefined number which represents the desired total number of rooms for a given floor.

Any paths which are marked open in the new room and link to an open path in an already existing room need to be changed to connected. This allows the check surround rooms function to identify the linked rooms. This is achieved through a simple "if" statement;

```
if (floorGrid[roomX][roomY].northPath == 1 && floorGrid[roomX][roomY + 1].southPath == 1) {
    floorGrid[roomX][roomY].northPath = 2;
    floorGrid[roomX][roomY + 1].southPath = 2;
}
```

Finally, the current room count is compared to the room max value. If enough rooms have still not be spawned the functions are repeated to place another room.

In practice however, this did operate as intended. This would just stop generating rooms when the maximum number had been reached, but it almost always left open paths that had not been matched. Instead an alternative version of the choose room function was created called *"CloseRooms"*. This function simply examines the four surrounding directions and chooses the specific room type which closes off all the available paths.

Now instead of comparing the room count to the room max, two different conditions could be used. The first would be when the room count plus the potential new room count is less than the room max. This would be the default scenario and the choose room function would be called. The alternative condition would be when the room count plus the potential new room count is greater than or equal to the room max. In this case, the close room function would be called instead to close up any remaining open paths;

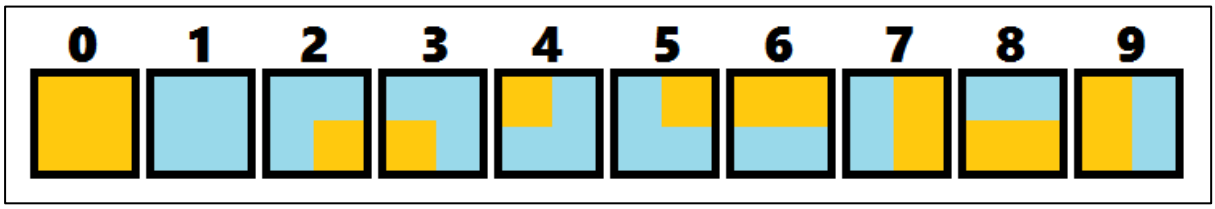
```
CheckSurroundingRooms(50, 50);

if (roomCount + potentialRooms < roomMax) {

    ChooseRoom();
}

if (roomCount + potentialRooms >= roomMax) {

    CloseRooms();
}
```

For example, in the case when the number of required rooms is 10, the current room count is at 7 and the potential room count is at 3. If rooms are placed in the 3 remaining potential positions, closing off all the open paths then the total number of rooms when generation stops will be the required 10.

These functions outline the procedural generation process and once they have completed the floor grid will be populated with all the relevant information about each room. From this point the level generator can be included with the main DirectX program to start drawing these rooms to the screen using a tile engine.

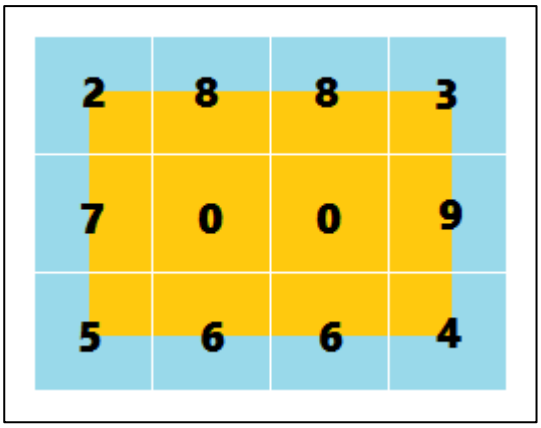
Tile Engine

The basic premise behind a tile engine, is to use a series of tiles to define a landscape rather than having to create a pre-defined image for every individual landscape. Consider the following 10 tiles;



10 example tiles for a tile engine

If the yellow represents land and the blue represents water, then these tiles could be positioned in multiple different combinations to create different islands. For example;



Example "island" designed using a tile engine

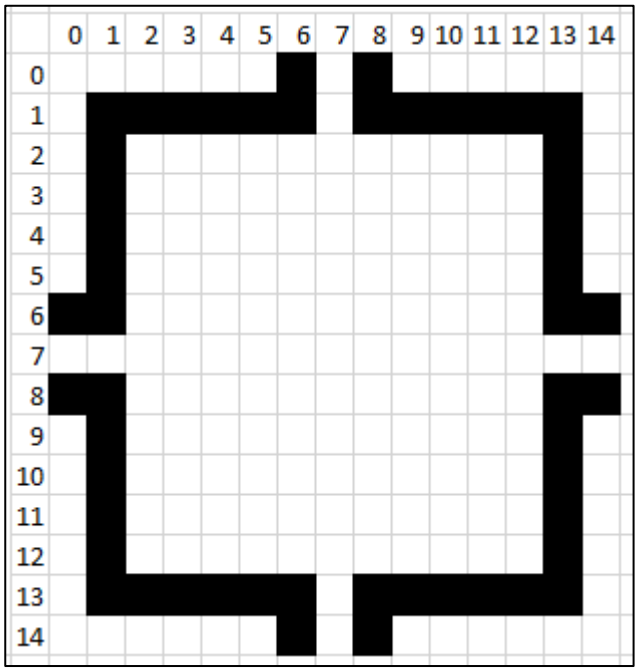
This means that all that is needed to describe a map is a two dimensional array of integers, with each integer referencing a particular tile. These tiles then just need to be drawn to the screen in the position defined by the array in order to recreate the entire map. Care does have to be taken to make sure that the edge of each tile will match up perfectly with any intended connecting tiles. Otherwise rough edges and obvious repeating patterns are created.

For my tile engine I started by defining a few parameters;

- *"TS"* is the height/width of a tile in pixels (all the tiles are square so they fit into a grid evenly)
- *"ROOMSIZE"* is the amount of tiles in the height/width of a room
- *"MW"* and *"MH"* are the map width and map height and represent the total number of tiles within the tile map. The value is defined as *"ROOMSIZE"* multiplied by the floor grid size.

A two dimensional integer array called *"map"* is declared to hold the tile map, with the size parameters being *"MW"* and *"MH"*.

The width/height of a room needed to be odd so that an entrance would line up with the centre of the room, and it also needed to be large enough so that different room structures could be placed inside. A good value seemed to be 15. In order to test different layouts, I used Excel to map out the desired shape of a room;

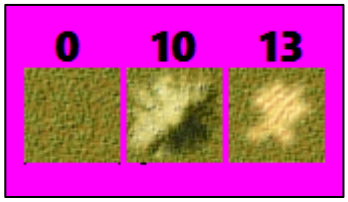


Excel representation of a room defined using tiles

Finally, I used a provided set of basic tiles to build a prototype to see how generating the rooms would look;

Strip of 16 prototype tiles

I used three of these tiles in my prototype; 0 to represent the floor, 10 represent a wall and 13 to represent the void space between rooms.



Three prototype tiles

Draw Room Function

A new function was needed which could examine a position on the floor grid to then populate the tile map with the tiles defining the room. This function is called *"DrawRoom"* and has three parameters; the room type *"rT"*, and the rooms floor grid co-ordinates *"xRoom"* and *"yRoom"*.

The most efficient way to handle this was to define the outline of the room first with no paths. Then depending on the room type the paths could be added by replacing the relevant positions on the map.

The main body of the room is populated using two loops. The outer loop covers all the y tile co-ordinates and the inner loop covers all the x tile co-ordinates. From this the appropriate tiles can be positioned as based on the layout I had designed in Excel.

Once the main walls of the room have been added to the tile map, the possible paths have to be considered. For example, a north path is need for any room with a room type of 0, 3, 6, 8, 10, 11, 12 and 14. For any of these room types; the tile map array is updated to add a path at the top of the room, the code for this appears as;

```
if (rT == 0 || rT == 3 || rT == 6 || rT == 8 || rT == 10 || rT == 11 || rT == 12 || rT == 14) { //north door room types

    map[(xRoom * ROOMSIZE) + (int)(ROOMSIZE / 2) - 1][(yRoom * ROOMSIZE)] = 10;
    map[(xRoom * ROOMSIZE) + (int)(ROOMSIZE / 2) + 1][(yRoom * ROOMSIZE)] = 10;
    map[(xRoom * ROOMSIZE) + (int)(ROOMSIZE / 2)][(yRoom * ROOMSIZE)] = 0;
    map[(xRoom * ROOMSIZE) + (int)(ROOMSIZE / 2)][(yRoom * ROOMSIZE) + 1] = 0;
}
```

The top left tile of a given room can be defined as;

$$(\text{“}xRoom\text{”} * \text{“ROOMSIZE”}), (\text{“}yRoom\text{”} * \text{“}ROOMSIZE\text{”})$$

This will give the give the exact amount of tiles that defines that position on the tile map at and makes it impossible for rooms to end up overlapping.

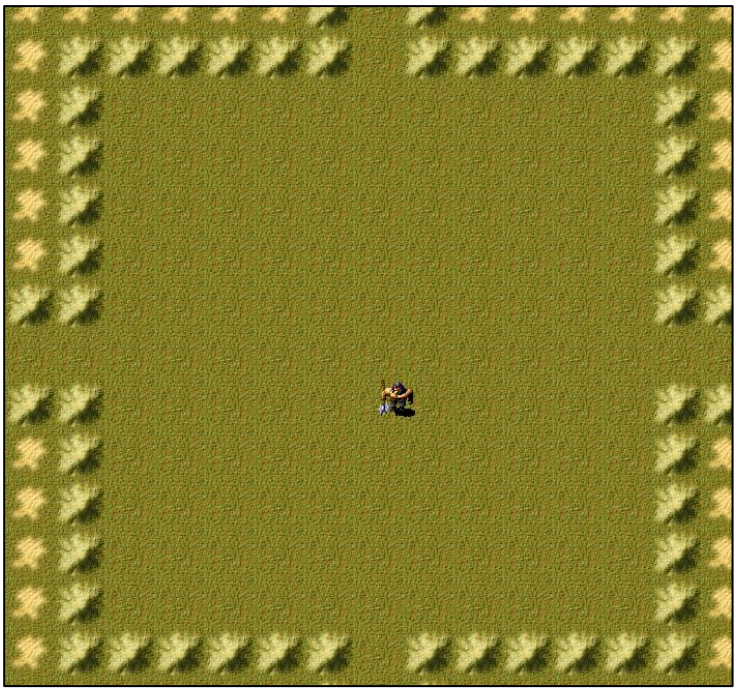
When it comes to actually drawing the tiles to the screen two nested loops are used to check every position within the tile map and then those values are drawn using the DirectX *“BltFast”* function. This function is called every render frame and requires 5 parameters;

- The x co-ordinate, in screen relative space, where the object will be drawn
- The y co-ordinate, in screen relative space, where the object will be drawn
- The direct draw surface which contains the source image
- A pointer to a DirectX *“RECT”* which contains the position of the required object within the source image
- A colour key definition to declare what colour should be made transparent

The x and y co-ordinates can be defined as the loop counter multiplied by the tile size. The direct draw surface will always be the earlier defined off screen surface of *“DD1”*. For each pass of the inner loop the RECT values need to be modified so that they refer to the correct tile within our source image. For example, to reference tile 10; the left position of the RECT needs to be (tile size times tile number) and right of the RECT needs to be ((tile size times tile number) plus tile size). As all the tiles are in the same line the y co-ordinates will be constant throughout. Setting this RECT appears as;

```
z = map[i][j];    //get map tile number
r.left  =TS*z;
r.top   =0;
r.right =TS*z+TS;
r.bottom=TS;
```

All that remains is to add a call to this function every time a new room is created using the spawn room function. That way the spawn room function will also be populating the tile map as well as setting the values in the floor grid. The prototype generator now produces rooms successfully;



Prototype room generator showing a room type 0

Screen Scrolling

To handle transitioning from one room to another, I wanted the camera to pan across to the next room and then become stationary once centred on the new room. The trigger for this would be when the player walks into one of the connecting paths in the current room. In order to implement this, I first needed to create the player character within the game.

A simple player was defined using a struct, which held the player's x and y co-ordinates and the x and y co-ordinates of the current room the player is occupying. The starting co-ordinates for the player would always be the centre of the starting room and players initial room co-ordinates would be that room (50, 50).

Another issue with using the tile grid is that an object can't be drawn to the screen if any part of it would appear outside of the defined screen area. This causes a memory failure and most often results in the program locking up. Handling this issue and addressing the screen scrolling are all part of the same problem.

Three sets of variables are required in order to manage screen scrolling;

- *"ssx"* and *"ssy"* to hold the co-ordinates of the top left pixel of the screen area in the world space. The screen can then be drawn between *"ssx"* and *"ssx + screenwidth"* as well as; *"ssy" and "ssy + screenheight".*
- *"sxmod"* and *"symod"* to hold the current screen co-ordinates modded by the tile size. This can be used to calculate the offset between a tile and the edge of the viable screen.
- *"sxnum"* and *"synum"* to hold the current number of whole tiles which will fit between the top left of the map and the current screen co-ordinates. These values are calculated by taking the current screen co-ordinate, subtracting the offset and then dividing by the tile size.

These variables can initially be used to make sure that all the tiles are drawn within the viable screen area as defined by screen width *"SW"* and screen height *"SH".* When drawing the tiles from the tile map; the current *"sxnum"* and *"synum"* values needed to be added to the loop values so that the corresponding values are drawn to the screen based on the cameras current position.

Also, when using the "*BltFast"* function to define the position to draw the tiles; the *"sxmod"* and *"symod"* values need to be subtracted from the drawing co-ordinates. This guarantees that the top left tile on the screen will always be drawn pixel perfectly into the top left corner, which also means that there will be a gap on the bottom and right of the screen the size of the offset value. This prevents any tiles being drawn out of the screen boundaries.

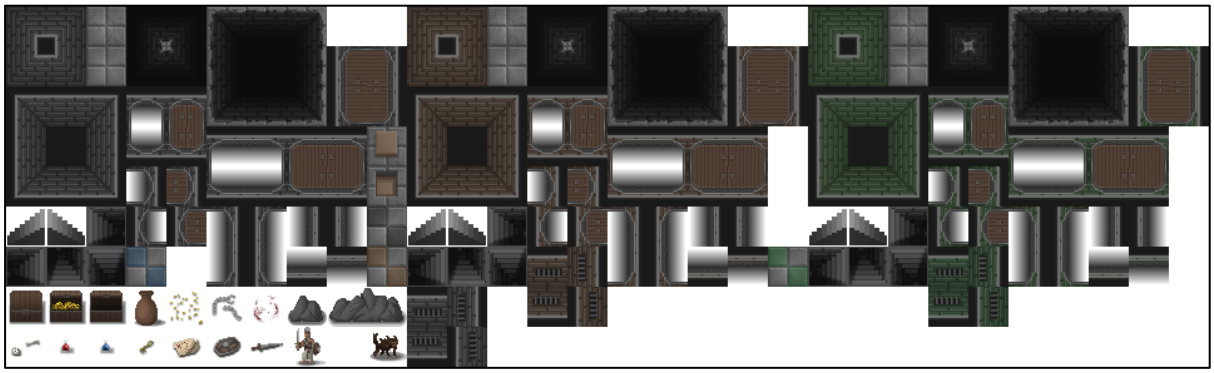
One issue this does create is a slight stuttering effect as the screen is moving. This is because as the space for a new tile to fit on the screen becomes available a tile instantly appears whilst a tile on the opposite side of the screen disappears. As a fix for this I used the "*BltFast"* function to draw black bars over the edge tiles in all directions to hide this effect.

In terms of handling the screen scrolling two more variables were required; *"destinationX"* and *"destinationY"*. When the player walks to the boundary of the room and triggers the screen scrolling, these variables are set to the top left pixel of the destination room. Each frame the destination values are compared to the current screen co-ordinates and if they don't match then the screen slowly increases or decreases its current co-ordinates to move closer to the destination. Once the destination has been reached (or overshot), the current co-ordinates are set to the destination co-ordinates. This last step is done to make sure the transitions are pixel perfect, otherwise the screen could start getting offset from the rooms as more transitions are made.

This also led to the creation of a new global variable *"loading"*. This variable is set to 0 during normal gameplay and is then set to 1 whilst the game is loading or transitioning. This can then be used to limit user input to only be read whilst the game is not loading.

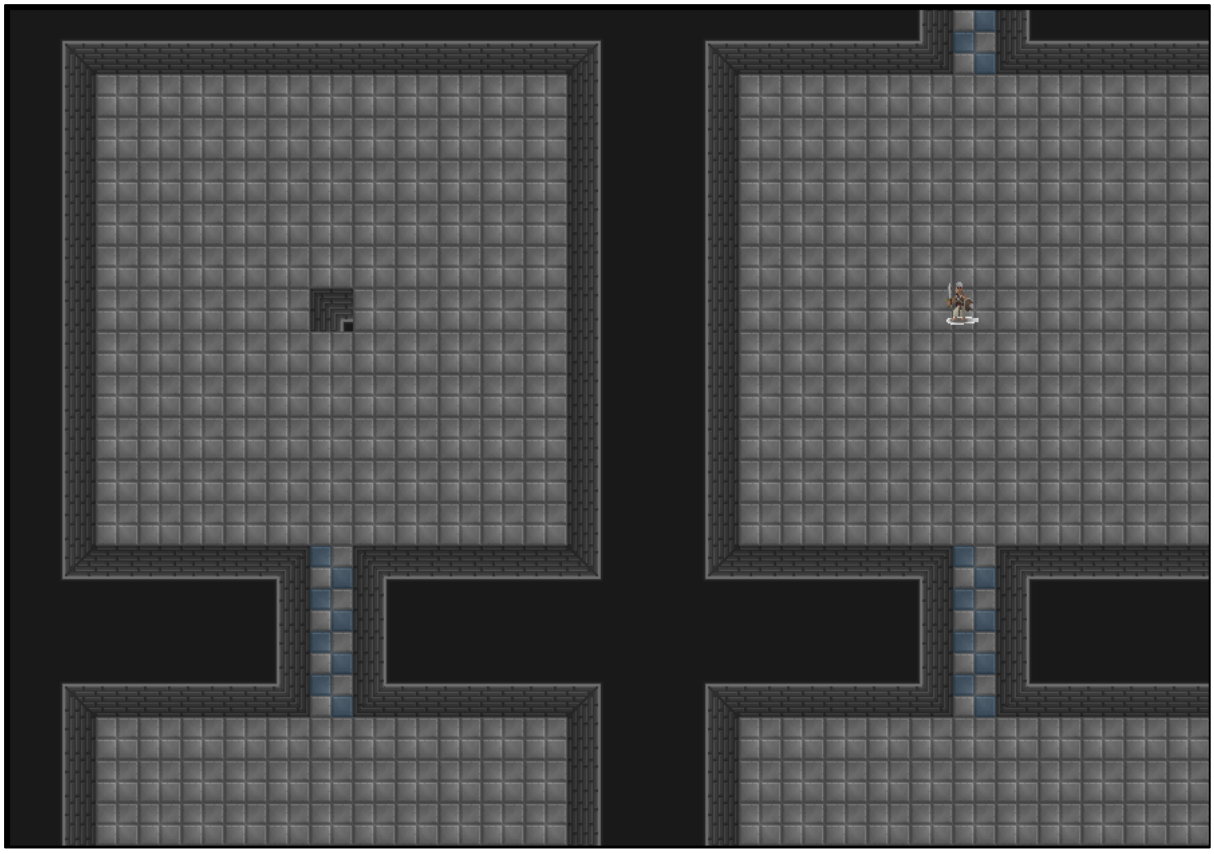
Final Tile Artwork

Now that the barebones of the game engine has been created, it seemed a good time to source some artwork which could be used to better represent the dungeon. I found an artist who was selling a 64-pixel dungeon tile set which seemed to fit the intended design well. It contained a series of different walls, floor tiles, pits, doors etc.;



Dungeon Tile Set

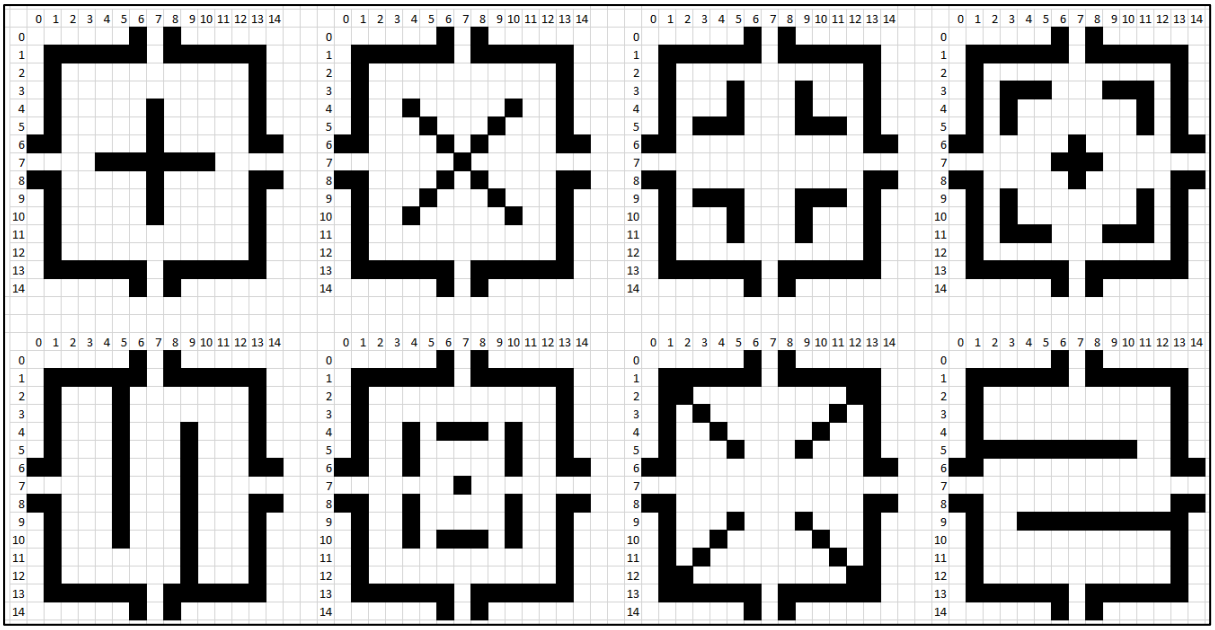
By making some small changes to the draw room function; the prototype tiles were replaced with the final dungeon tile set. This added a little complexity as there were now 12 different wall tiles representing the different orientations of walls and wall corners. I also chose to temporarily change the player sprite to provided character. After I had changed the *"TS"* value to match the new tile pixel size, the dungeon looked like;



Generated Dungeon using the final tile set

Structuring Rooms

The internals of each room also need to be procedurally generated, however the walls created inside the room can't block any potential exits as any given room could have a path at the centre of any wall. The best solution for this was to define some pre-defined room shapes which would be compatible with all possible room types. I used the Excel template I had created earlier to mark out 8 different room structures;



The 8 potential room structures

A new integer variable was added to the room struct called *"roomStructure".* This variable simply defines which of the above internal structures is used for a room. It is defined randomly when the room is first created. The only rooms which don't have one of these structures are the starting room, key rooms and the exit room, which will all need bespoke structures.

A function called *"StructureRoom"* handles setting the tile map values for the internal wall positions in a similar way to how the draw room function sets the outer walls. An example of the code to set the tile map for the first internal structure the "lateral cross" appears as;

```
if (floorGrid[xRoom][yRoom].roomStructure == 1) { // lateral centre cross

    for (int i = 2; i < ROOMSIZE - 2; i++) { //double loop covering only the floor space of a room
        for (int j = 2; j < ROOMSIZE - 2; j++) {

            if (i >= 4 && i <= 10 && j == 7) { map[(xRoom * ROOMSIZE) + j][(yRoom * ROOMSIZE) + i] = 18; }
            if (j >= 4 && j <= 10 && i == 7) { map[(xRoom * ROOMSIZE) + j][(yRoom * ROOMSIZE) + i] = 18; }
        }
    }
}
```

Mini-Map

Drawing the mini-map shouldn't be too difficult as the data needed is already stored within the floor grid, and the tile engine can be used to place tiles of the different room types next to each in order to draw the mini-map to the screen. It would also be useful to show the player which rooms they have previously visited to help prevent unnecessary backtracking. In order to track this; a new variable was added to the room struct; *"visited"* which is set to true when the player enters the room for the first time. If a map position is visited then it can be drawn on the mini-map as a white room, and if it is unvisited it can be displayed as a grey room, however this does require two sets of sprites.

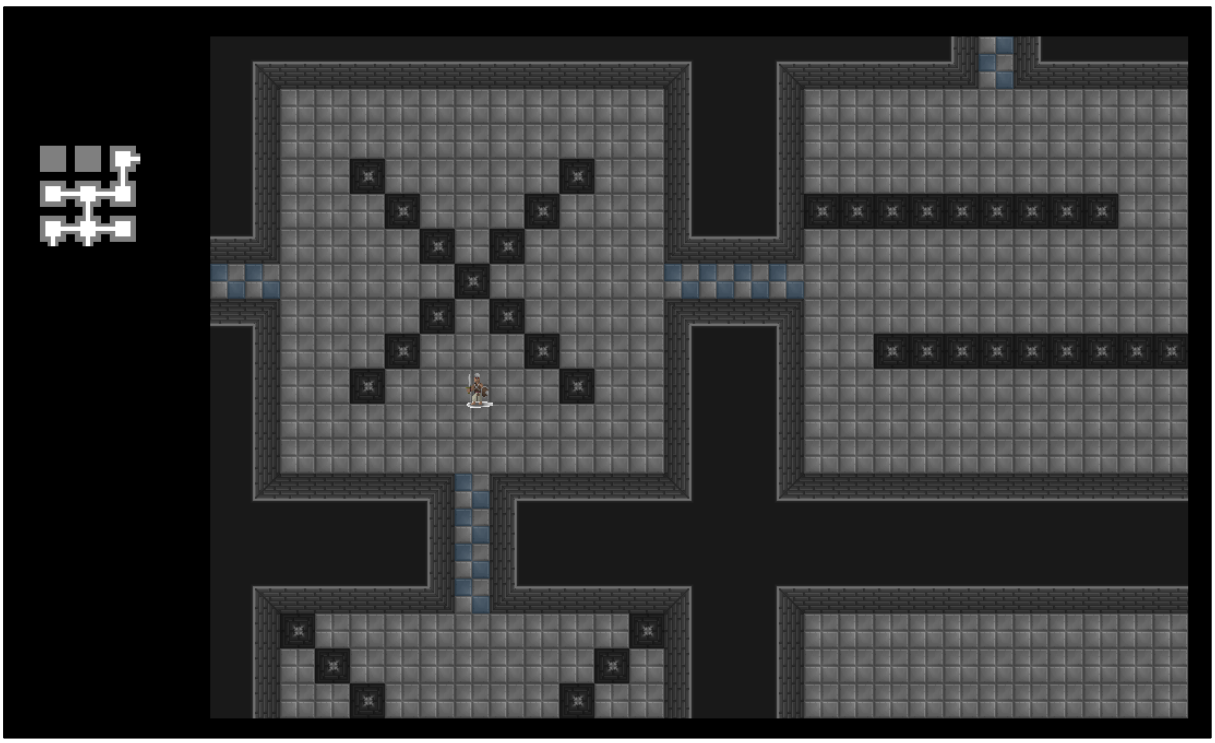
The mini-map will only display 9 rooms with the centre room being the one currently occupied by the player. This means that by using two nested loops which start at -1 and run to 1, all 9 positions can be inspected. For example, the first run through the loops will have them both with a value of -1. If the floor grid is examined at the player's current position but minus 1 in both the x and

y axis it will be looking at the room north-west of the player. By the time both loops have finished all the surrounding rooms and the players current room will have been examined.

Within the inner loop a new variable is declared called *"roomType"*, which is initially set to -1 to represent an empty room. If the current room being examined is occupied; then *"roomType"* is set to the examined room's room type, otherwise the value remains -1, to show that it is an empty space.

All of the mini-map images are drawn using the *"BltFast"* function, and placing them on the tile grid on top of the black spaces used to border the main game window. For a room type of -1 a grey square is drawn to represent an empty space, for any other value the appropriate room type image is selected (either a white one for a visited room or a grey one for an un-visited room) and drawn to the correct position.

The game with room structures and the mini-map now appears as;

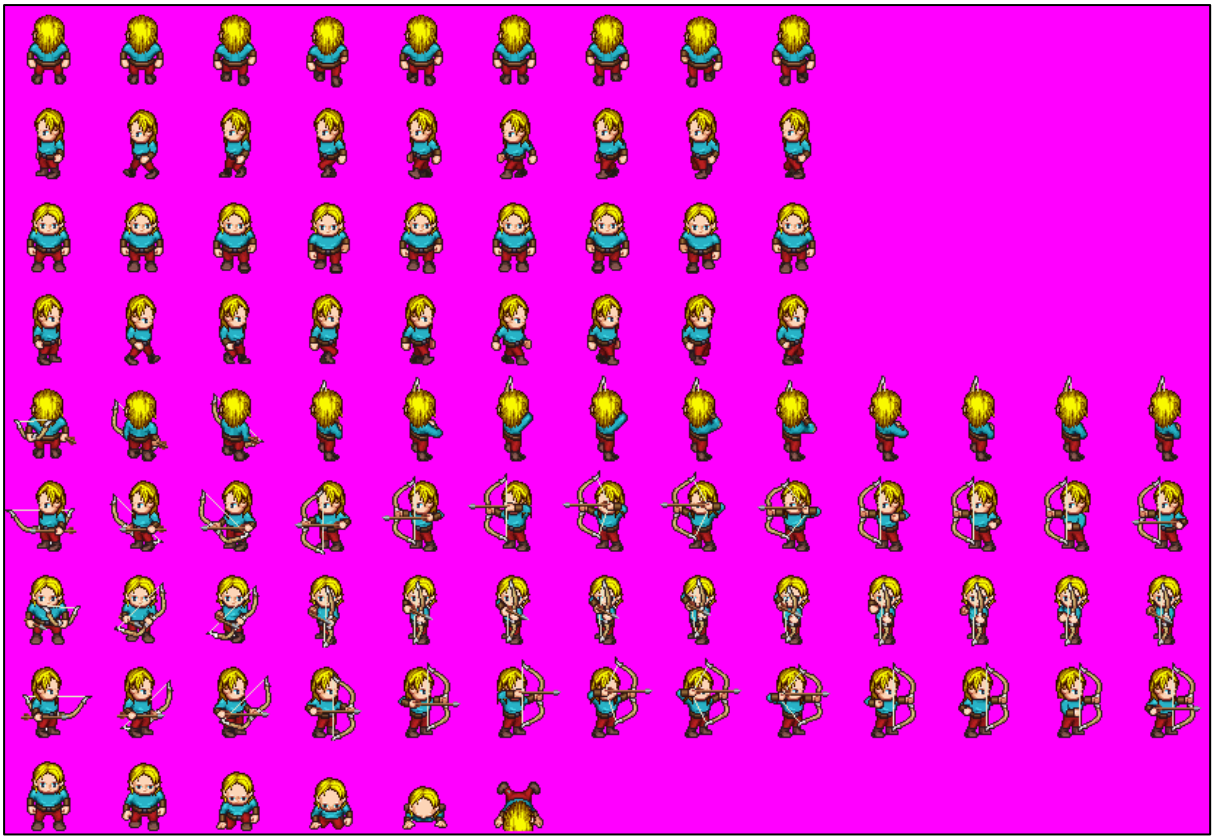


Dungeon with generated room structures and mini-map

Player Animation and Collision

Animation

The next step was to find some animated sprites which could be used to represent the player, ideally containing animations for; walking in four directions and shooting an arrow from a bow in four directions. I came across a website which could produce a character sprite sheet from user input values. This allowed be to customise my player character, choose the clothes they wear, and even the weapon they use. This provided me with far more sprites than I required so I cut it down to just include the relevant walking, shooting and dying sprites. I also made sure that the background colour was changed to match the transparency colour being used by the game; (255,0,255) pink. These sprites were also 64-pixels square, which allowed them to be easily inserted into the existing tile sheet. The sprite sheet appears as;



Character animation sprites for walking, shooting and dying

In order to keep track of the current player animation; a new integer variable needs to be added to the player struct called *"animationState"*. This variable is used to represent the current animation state of the player; 0 – 3 are the standing animations (NWSE), 4 – 7 are the walking animations (NWSE), 8 – 11 are the shooting animations (NWSE) and 12 is the dying animation.

The animation state is decided by the current user input. For example, if the user is holding the button to walk right, then the animation state is changed to 7 (walking east) and when the user releases the button and the player becomes stationary the animation state is changed to 3 (standing east).

```
if (!KEY_DOWN(0x57) && player.animationState == 4) { f = 0; player.animationState = 0; }
if (!KEY_DOWN(0x53) && player.animationState == 6) { f = 0; player.animationState = 2; }
if (!KEY_DOWN(0x41) && player.animationState == 5) { f = 0; player.animationState = 1; }
if (!KEY_DOWN(0x44) && player.animationState == 7) { f = 0; player.animationState = 3; }
```

Drawing the current animation state to the screen requires a few new variables;

- *"maxf"* is one less than the maximum number of frames for a given animation. For example; the walking animation has 9 frames, so the maxf value for that animation would be 8
- *"f"* is the current frame of the animation. This is used to keep track of which sprite needs to be drawn to the screen.
- *"delay"* is the time in milliseconds to be waited between each frame of the animation
- *"t1"* and *"t2"* store tick counts used to check if the required time has passed to draw the next frame.

When it comes to drawing the player, first the animation state is considered by the *"AnimationSelector"* function. Depending on the animation state; the maximum frame and delay

values are changed to match the frame count of the given animation and an appropriate delay between frames. The "RECT" used for the *"BltFast"* function is also defined for the current animation state. For the standing position no actual animation is needed as there is only a single frame, however for the other animations the current frame count is used to decided which sprite from the sprite sheet should be drawn to the screen next. Here is an example of the code for when the player's animation state is walking north;

```
if (player.animationState == 4) {

    maxf = 8;
    delay = 50;
    r.left = f * TS;
    r.top = TS * 2;
    r.right = TS + (f * TS);
    r.bottom = (TS * 2) + TS;
}
```

Notice how the left and right values of the "RECT" are variable based on the current frame count. For example, if the frame count was 0 then the left edge of the sprite, as located on the off screen surface, would be at 0, and the right edge would be at 64 (given that the current tile size is 64).

In order to update the current frame count, a comparison is made between the two stored tick values. *"t1"* gets the tick count at the programs initialisation and *"t2"* gets the tick count once per frame. The amount of time passed, in milliseconds, between recording these values can be calculated by subtracting *"t1"* from *"t2".* If that amount is greater than the delay between frames then *"t1"* is set to the current *"t2"* value, essentially resetting the timer. The frame count is then incremented by one, so that the next frame can be drawn. If the frame count is incremented above the maximum amount of frames, then it set back to 0 so that animation can loop again. This process is coded as;

```
// Decide which frame will be blitted next
t2 = GetTickCount();
if ((t2 - t1) > delay) {
    t1 = t2;
    f++;
    if (f > maxf) f = 0;
}
```

Now all the relevant data has been gathered to draw the animated player to the screen using the *"BltFast"* function;

```
DDB->BltFast(player.xpos - ssx, player.ypos - ssy, DD1, &r, DDBLTFAST_SRCCOLORKEY);
```

It is important to note that when using *"BltFast"* that the position the object is chosen to be drawn to is relative to the screen space, not the world space. As this is the case; the player can't just be drawn at their current co-ordinates, but instead must be drawn at their current co-ordinates minus the current screen scroll amount. This makes the player drawn relative to the screen, even though the co-ordinates of the player are relative to the world. The same off screen surface is used within this function; as the player sprites have been added to the main bitmap. The "RECT" has already been defined depending on the animation state and current frame using the steps mentioned above.

It is also important to make sure that the current frame count is reset to 0 every time the animation state changes, otherwise the new animation could start with the wrong frame. For example, assume the player was walking and was currently on frame 8 of the animation, if they then tried to shoot the bow the program would begin drawing it from the 8th frame, which would instantly show the bow drawn and ready to fire, missing out all the previous animation frames. The player should also not be drawn if they are outside the screen area, although this should be impossible due to the nature of the rooms, it is worth using a condition to avoid any potential crashes. The player is only drawn if the following condition is met;

```
if (player.xpos > ssx && player.xpos < ssx + SW && player.ypos > ssy && player.ypos < ssy + SH) {
```

At this point the player is animated whilst standing and walking in every direction, with the infrastructure created to handle the death and shooting animations when the necessary code is added to the game.

Wall Collision

The next functionality to add to the player was collisions. At this point in the game the only object for the player to collide with is the walls. There are multiple different ways to handle collisions, but as the walls are stationary tiles, a simple solution can be applied using the tile map.

First two new variables are needed in the player struct; *"tileX"* and *"tileY"*. These values represent the location of the tile the player is currently standing on and can calculated by dividing the player co-ordinates by the tile size. By examining the value stored in the tile map at the player's current position it can then be determined whether a collision is occurring. For example, if the value of the tile the player was standing on was equal to a wall tile, then a collision can occur.

When the player is moving up, the tile co-ordinates need to be determined from the top of the players bounding box, whereas when the player is moving down, the tile co-ordinates need to be derived from the bottom of the player bounding box. This rule is also true for moving left and right;

Moving Up

```
player.tileX = (player.xpos) / TS;
player.tileY = (player.ypos) / TS;
```

Moving Down

```
player.tileX = (player.xpos) / TS;
player.tileY = (player.ypos + player.height) / TS;
```

Moving Left

```
player.tileX = (player.xpos) / TS;
player.tileY = (player.ypos) / TS;
```

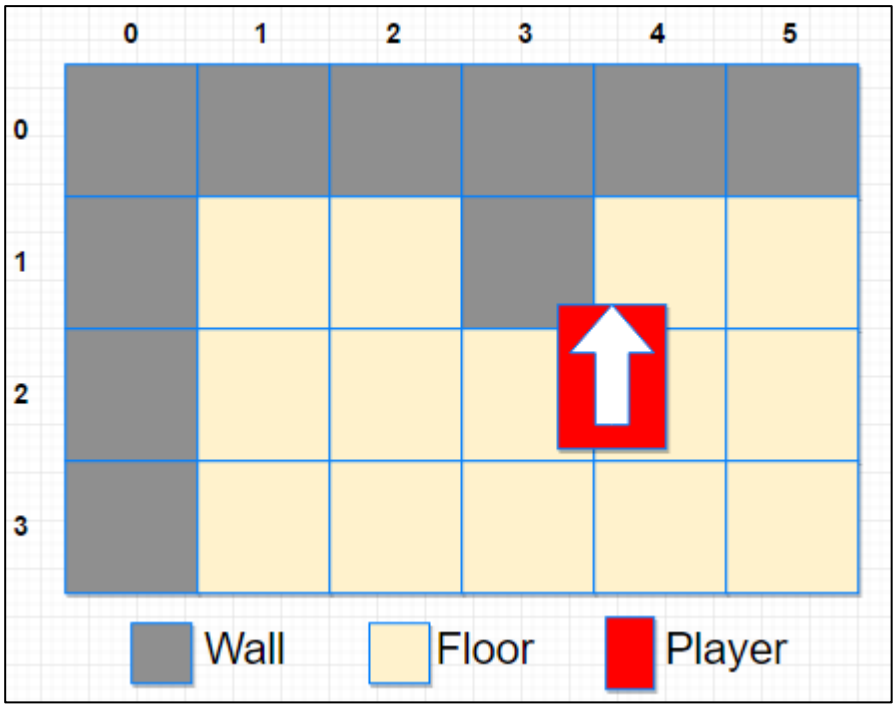
Moving Right

```
player.tileX = (player.xpos + player.width ) / TS;
player.tileY = (player.ypos) / TS;
```

If the default player co-ordinates were always used to set the tile position, then when the player is moving down they would clip through the wall until the top of their bounding box collided. Two static constants were also added to the player struct to help with this issue, they define the player's width and height in pixels and can be used to easily and consistently define the bounding boxes for collisions.

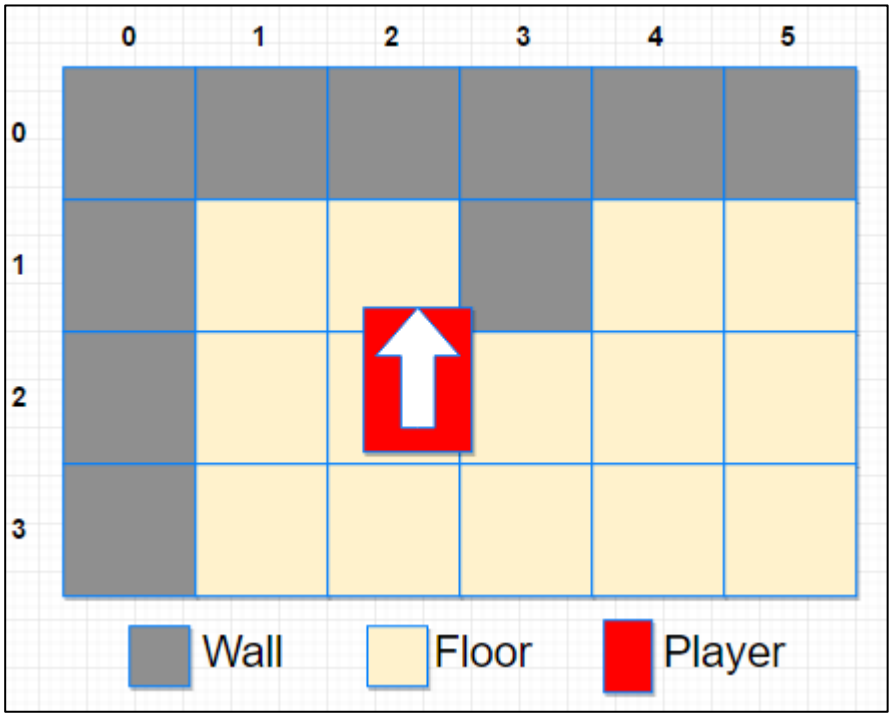
The player's tile positions are updated every render frame, using the appropriate definition shown above, depending on the direction the player is moving. The players current tile can then be compared against known wall tiles to check if a collision is occurring.

As the tiles, and the players bounding box are parallel to either the x or y axis, only two points of the players bounding box need to be checked for a collision when the player is moving. Consider the following example;



Wall collision example 1

In the above scenario the player is moving up, so the tile position is set the tile located at the player's x and y co-ordinates; tile (3,1). In this case a collision would be detected because that tile is marked as a wall. However, consider the player was on the other side of this wall;

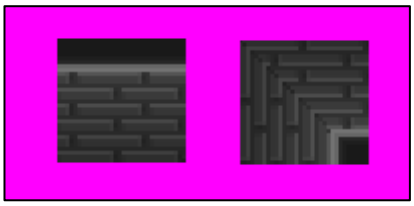


Wall collision example 2

If the players tile was simply derived from the x and y co-ordinates, then it would be tile (2,1) which the player would not collide with because it is marked as the floor. To collide correctly the top right of the player's bounding box also needs to be examined, the x co-ordinate of this tile could be defined as *"(player.xpos + player.width)/TS"*. The tile at this location would be the wall tile (3,1) which would collide with the player.

This shows that two corners of the players bounding box need to be examined for each direction that the player can move. A collision occurs if either of these corners detects a wall tile.

As the tile set being used contains different wall sprites for each direction, only certain walls need to be considered for collision based on the direction the player is moving. For example, consider these two tiles;

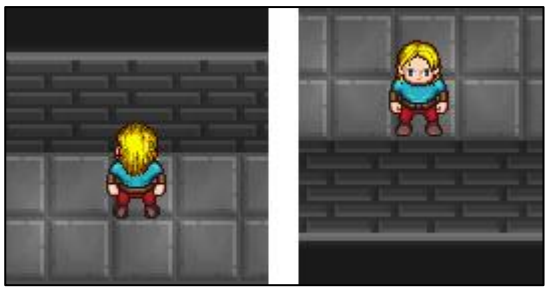


Wall tiles

The first tile will only ever been found on the wall at the top of a room, this means that the player will only come into contact with it when they are moving up. The second tile is an outer corner which the player will only come into contact with when moving down or right. This means that collisions for these tiles only need to be checked if the player is moving in a direction where they can possibly come into contact with the tile. I considered each wall tile that was used in creating a room and compiled a list of which walls the player could touch for each direction they were moving.

In the event that a collision is detected; the player needs to be have their position altered by the player walk speed, in the opposite direction to which they are travelling. As collisions and player inputs are calculated every render frame, what happens is that the player moves inside the wall, the collision is detected and then the player is returned to their previous position. As the player is drawn after both of these calculations, to the user, the player appears to not move at all when a collision occurs.

This collision method also allowed me to vary how the player bounding box is handled depending on the direction they are moving. In order to give the room perspective, I wanted the top half of the player sprite to overlap the wall when moving up, whilst when moving down I wanted the collision to occur as soon as the bottom of the player touches the wall. To achieve this; when the player is moving up I define the *"tileY"* value using the player's y position plus 30. This means that when moving up, the players bounding box is 30 pixels below the top of the player, allowing for a slight overlap with walls;



Forced perspective by varying player's bounding box

Combining all of these features; the code for collision detection and player movement when the player is moving up appears as;

```
if (KEY_DOWN(0x57)) { //move up

    player.animationState = 4;

    player.ypos -= walkSpeed;
    player.tileX = (player.xpos + 20) / TS;
    player.tileY = (player.ypos + 30) / TS;
    // wall collisions
    if (map[player.tileX][player.tileY] == 7 || map[(player.xpos + player.width) / TS][player.tileY] == 7) player.ypos += walkSpeed;
    else if (map[player.tileX][player.tileY] == 8 || map[(player.xpos + player.width) / TS][player.tileY] == 8) player.ypos += walkSpeed;
    else if (map[player.tileX][player.tileY] == 9 || map[(player.xpos + player.width) / TS][player.tileY] == 9) player.ypos += walkSpeed;
    else if (map[player.tileX][player.tileY] == 14 || map[(player.xpos + player.width) / TS][player.tileY] == 14) player.ypos += walkSpeed;
    else if (map[player.tileX][player.tileY] == 15 || map[(player.xpos + player.width) / TS][player.tileY] == 15) player.ypos += walkSpeed;
    else if (map[player.tileX][player.tileY] == 18 || map[(player.xpos + player.width) / TS][player.tileY] == 18) player.ypos += walkSpeed;
}
```

Level Progression

As mentioned in the specification, the way to advance to the next level is to collect the necessary amount of keys required to open the exit. The amount of keys needed to open the exit varies depending on the amount of rooms in the current floor, the formula is;

$$Number\ of\ Keys = 1 + \left(\frac{Number\ of\ Rooms}{10}\right)$$

The number of keys is also stored as an integer, so until a new multiple of 10 rooms has been reached the number does not increase. For example, if there were 19 rooms in a floor the number of keys would be 2, as the integer data type simply removes any decimal information. The program uses this formula to populate the integer variable *"keyRooms"* which tells the level generator how many key rooms need to be designated per floor.

There needs to be a different room for each key on a given floor, and a single exit room. The path in the exit room will initially appear blocked until all the floors keys have been collected by the player. These rooms are already defined by the room generator and can be identified as rooms with *"roomContents"* values of 2 or 3.

Bespoke internal room structures were also created for these two rooms so that they could be easily recognisable to the player. The key room simply has a coloured tile in the centre for the key to be located and the exit room uses a downward staircase sprite surrounded by wall tiles.

The keys are defined using a struct which holds a variable for whether the key is active, and two variables for the keys x and y co-ordinates. An array of 100 keys is then declared, to cover the maximum possible number of keys if every room in the floor grid was populated.

When the level generator designates a room to be the key room, it uses a loop to find the first inactive key in the array. It then sets that key to active and sets the co-ordinates to the centre tile in the room. Within the draw room function, the definition for a key room appears as;

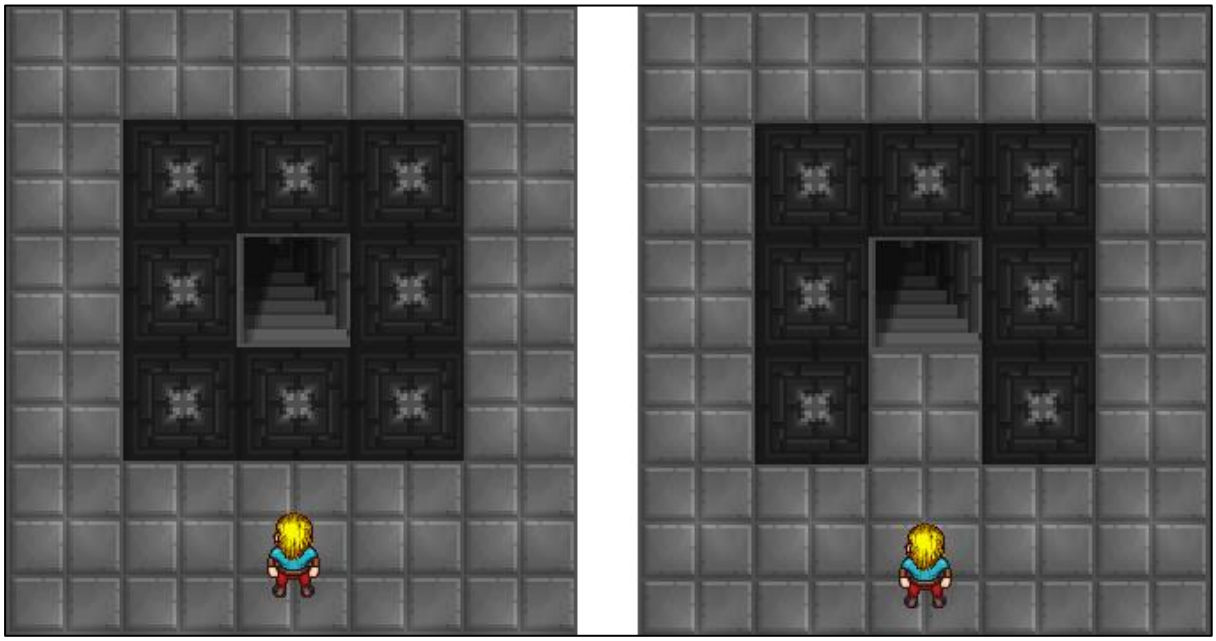
```
if (floorGrid[xRoom][yRoom].roomContents == 2) {
    map[(xRoom * ROOMSIZE) + (int)(ROOMSIZE / 2)][(yRoom * ROOMSIZE) + (int)(ROOMSIZE / 2)] = 28;

    for (int i = 0; i < 100; i++) {

        if (key[i].active == 0) {
            key[i].active = 1;
            key[i].xpos = (xRoom * ROOMSIZE * TS) + (int)((ROOMSIZE / 2) * TS);
            key[i].ypos = (yRoom * ROOMSIZE * TS) + (int)((ROOMSIZE / 2) * TS);
            break;
        }
    }
}
```

Drawing the key to the screen is done inside a loop which checks all of the positions in the key array. A key is only drawn if it is active and it is within the screen boundaries. The keys position on the off screen surface can then be set using a constant *"RECT"* and then *"BltFast"* can be called. It is worth noting that the way DirectX draws to the screen is known as a painter's algorithm. This means that if two objects are drawn to the screen, the second object will be drawn layered above the first. This can be used to give appropriate perspective to different sprites. In this example, the key needs to be drawn above the tiles, but below the player, so that it appears the player can walk over the key.

The player also needs to collide with the key in order to collect it. As this collision does not need a high level of accuracy, a collision occurs simply when the player is on the same tile as the key. The players tile position is already determined by the direction they are walking, and the keys tile will be the centre tile of the key room. If these positions match a collision occurs; the key is set to inactive, has its co-ordinates set back to 0, and a new player variable *"keyCount"* is incremented by 1;

```
// key pickup
for (i = 0; i < 100; i++){
    if (player.tileX == key[i].xpos / TS && player.tileY == key[i].ypos / TS && key[i].active == 1) {

        key[i].active = 0;
        key[i].xpos = 0; key[i].ypos = 0;
        player.keyCount++;
    }
}
```

When the exit room is populated; the tile co-ordinates of the bottom middle wall blocking the exit is saved to two global variables; *"exitXpos"* and *"exitYpos"*. This is so that when the number of keys required to open the exit have been collected by the player, this position on the floor map can be changed from a wall to the floor, allowing the player to reach the exit.



(Left) Closed Exit, (Right) Open Exit

All that remains is to add the staircase tile to the player collision code, so that when the player touches the staircase a function can be called to advance to the next level;

```
else if (map[p.tileX][p.tileY] == 20 || map[(p.xpos + p.width + 12) / TS][p.tileY] == 20) AdvanceLevel();
```

When advancing to the next level most of the variables need to be reset to their initial values at the start of the game. The only variables which carry over between levels are the player's health and arrow count. Also for every new level, the *"roomMax"* count increases by 3, so that each floor has 3 more rooms than the last did. Once the floor grid, tile map, and player position data is reset, the level generator can be called again to generate a new floor with 3 more rooms than the last. All of these commands are stored within the function *"AdvanceLevel"* which can be called whenever the player collides with the staircase leading to the next floor.

Bow and Arrow Mechanics

Initially, a few new variables are needed to handle arrows. The player struct needs a new integer variable *"arrowCount"* to store the amount of arrows the player is holding. A new struct is also needed to describe an arrow, this struct contains the following variables;

```
struct Arrow
{
    static const int speed = 12, delay = 200;
    static const int width = 8, height = 33;
    int active; //0 = no arrow, 1 = arrow
    int state;  //1 = in flight, 2 = at rest
    int t1, t2, f;      //animation variables
    int tileX, tileY;   //arrows current tile position
    int xpos, ypos;     //arrows current pixel position
    int direction; // 0 - 4 (NWSE)
} arrow[100];
```

Most of these variables are similar to those needed to describe any object, such as the position, tile position. The *"state"* integer is set to 1 when the arrow is in flight and 2 once it has collided and come to rest. This is used to identify whether the arrow should be damaging an enemy or being collected by the player. The integer *"direction"* determines which direction the arrow travels when it is fired. This is determined by the direction the player is facing. Each arrow also has its own timing variables for animation. The arrow has no animation in flight, but to make it easier to identify when it is at rest, it flashes with a yellow border. An array of 100 arrows is then declared, these can be set to active once a player fires one and set back to inactive once it is collected by the player.

The action of the player firing the bow is based on the bow being drawn when the space bar is depressed and fired when the space bar is released. This is achieved by considering the current frame of the shooting animation when the space bar is released.

Sprites for shooting west animation

Initially, a check is performed to make sure the player has at least 1 arrow, otherwise the bow can't be fired. If this check is passed, then when the player presses the space bar, the animation

for drawing the bow begins. The current direction of the player determines which of the shooting animations is chosen;

```
if (KEY_DOWN(VK_SPACE) && (player.animationState == 5 || player.animationState == 1)) { f = 0; player.animationState = 9; }
```

If the space key remains depressed and the last frame drawn was frame 8 (which is the fully drawn bow), then the animation loops back to frame 7. This makes the player shake a little as the animation will just cycle between the 7th and 8th frames. This looks like the player holding the tension of the bow ready to fire.

```
if (KEY_DOWN(VK_SPACE) && player.animationState == 9 && f == 9) { f = 7; }
```

If the space bar is released before the 6th frame is reached, then no arrow is fired, as the bow was not considered drawn yet. Instead the player just returns to the standing animation for that direction.

```
if (!KEY_DOWN(VK_SPACE) && player.animationState == 9 && f < 6) { f = 0; player.animationState = 1; }
```

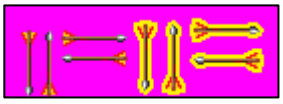
If the bow was fully drawn (the animation was between frames 6 and 8) when the space bar is released, then an arrow is fired. When an arrow is fired; a loop checks every position in the arrow array to discover the first inactive arrow. That arrow can then be set to active and given a state of 1, to represent that the arrow is now in flight. The direction of the arrow is set to match the direction the player is facing. The x and y position of the arrow are manually determined for each direction and line up with the tip of the bow

The players arrow count can now be decremented by 1 and the frame of the shooting animation can be set to 9 (the frame showing an empty bow after the arrow has been fired).

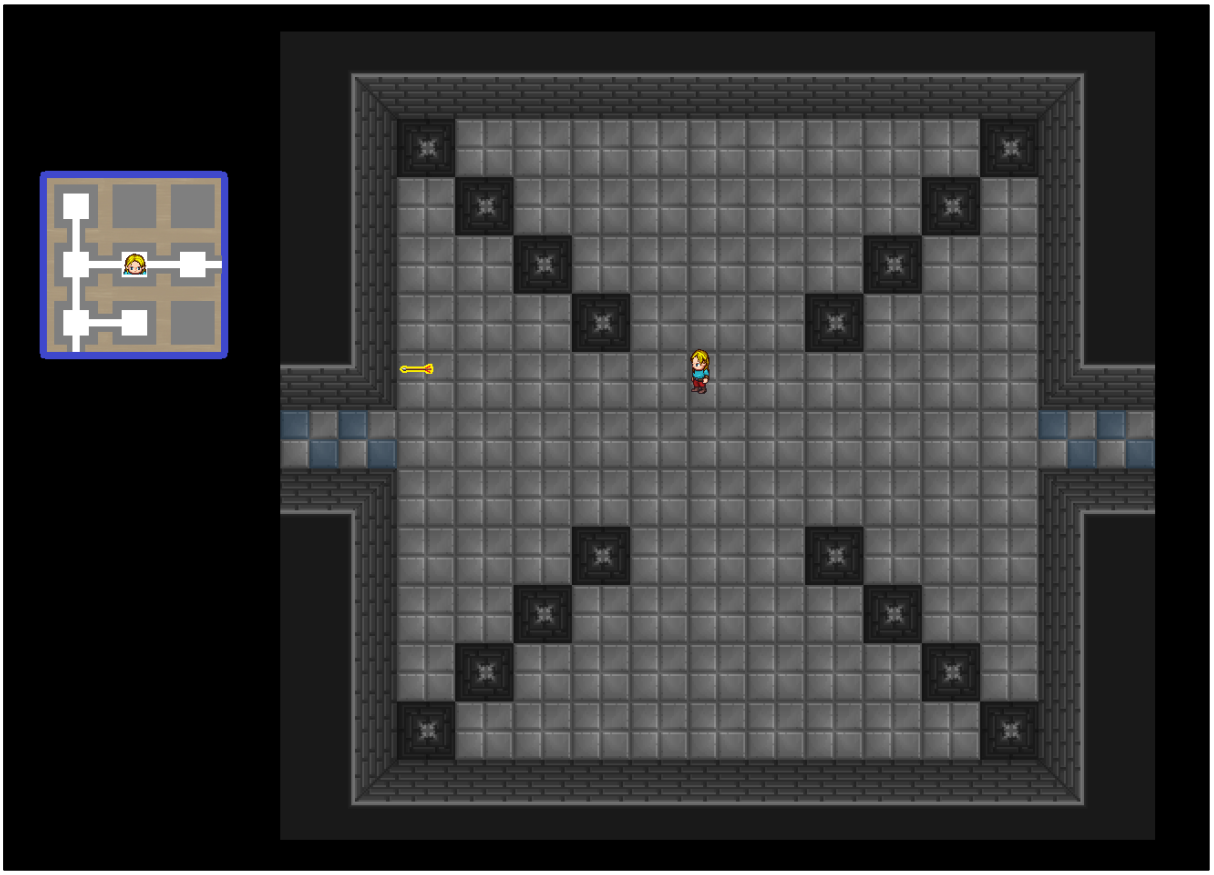
```
if (!KEY_DOWN(VK_SPACE) && player.animationState == 9 && (f == 6 || f == 7 || f == 8)) {

    for (i = 0; i < 100; i++) {
        if (arrow[i].active == 0) {
            arrow[i].active = 1; arrow[i].state = 1;
            arrow[i].direction = 1;
            arrow[i].xpos = player.xpos + 3;
            arrow[i].ypos = player.ypos + 29;
            player.arrowCount--;
            f = 9;
            break;
        }
    }
}
```

Once an arrow is in flight it moves at *"arrow.speed"* per frame, and it can collide with walls using the same collision detection method as the player. When a collision is recorded; the arrow stops moving and the state is changed to at rest. If the player collides with the arrow whilst it is at rest; it is picked up. This sets the arrow to inactive, resets the variables stored in the arrow struct, and increments the players arrow count by 1.

The arrow is animated whilst at rest, switching between two frames;



The delay between frames is set to 200 milliseconds, and this gives the arrow the appearance of slowly flashing with a yellow border. This makes the arrow easier to spot against the floor. As each arrow has its own timing variables, all arrows at rest don't flash simultaneously. With these new features the game screen now appears as;

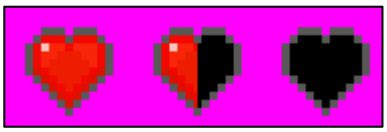


Game screen with animated character, collisions and shooting

Some other small improvements were made; the screen size and tile size were updated so that one room would fit better into a single screen. The mini-map also now has a background which is drawn prior to the mini-map rooms, and a marker to show the current room the player is in, which is drawn after the mini-map rooms.

Game UI

It would be useful to also display the player's health and arrow count on the screen. A new variable *"health"* is added to the player struct to store the current amount of health and another variable called *"healthMax"*, which represents the maximum health the player can have. Each 1 health will be displayed as a half heart using a "Zelda" style heart system;



UI health sprites

A "RECT" was defined for each of the heart sprites; *"hFull", "hHalf"* and *"hEmpty"*. The appropriate combination of these "RECTs" can then be drawn to the screen depending on the player's current health. The player would initially start with a maximum health of 6 and a current

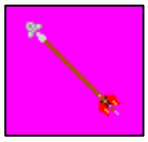
health value of 6. This means that three full hearts need to be displayed to represent this, so the "RECT" for all three heart positions needs to be *"hFull"* so that three full hearts are drawn;

```
if (player.health == 6) {
    DDB->BltFast(21 * TS, 4 * TS, DD1, &hFull, DDBLTFAST_SRCCOLORKEY);
    DDB->BltFast(22 * TS, 4 * TS, DD1, &hFull, DDBLTFAST_SRCCOLORKEY);
    DDB->BltFast(23 * TS, 4 * TS, DD1, &hFull, DDBLTFAST_SRCCOLORKEY);
}
```

Conditions are used to consider every possible value for player health, and then the appropriate "RECT" are used within the *"BltFast"* to draw the correct combination of hearts. For example, if the player had 3 health, they should have a full heart, a half heart and an empty heart;

```
if (player.health == 3) {
    DDB->BltFast(21 * TS, 4 * TS, DD1, &hFull, DDBLTFAST_SRCCOLORKEY);
    DDB->BltFast(22 * TS, 4 * TS, DD1, &hHalf, DDBLTFAST_SRCCOLORKEY);
    DDB->BltFast(23 * TS, 4 * TS, DD1, &hEmpty, DDBLTFAST_SRCCOLORKEY);
}
```

The arrow UI will just display an image of an arrow for each arrow that the player holds, the sprite for this arrow is;



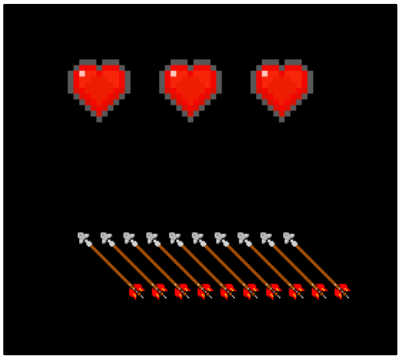
UI Arrow Sprite

This is achieved using a loop which runs between 0 and the players arrow count. For every run of the loop the next arrow is drawn one quarter of a tile to the right of the last. The implementation for this is done within the x drawing co-ordinate in a *"BltFast"* function;

```
for (i = 0; i < player.arrowCount; i++) {

    DDB->BltFast(21.2 * TS + ((TS/4) * i), 6 * TS, DD1, &r, DDBLTFAST_SRCCOLORKEY);
}
```

A variable for *"arrowMax"* was also added to the player struct to limit the maximum number of arrows the player can hold at one time.

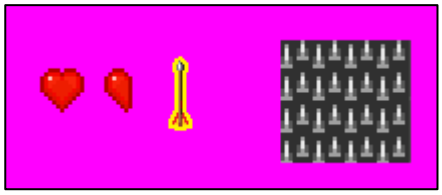
When the player has 6 health and is holding 10 arrows the in game UI appears as;



In-game UI

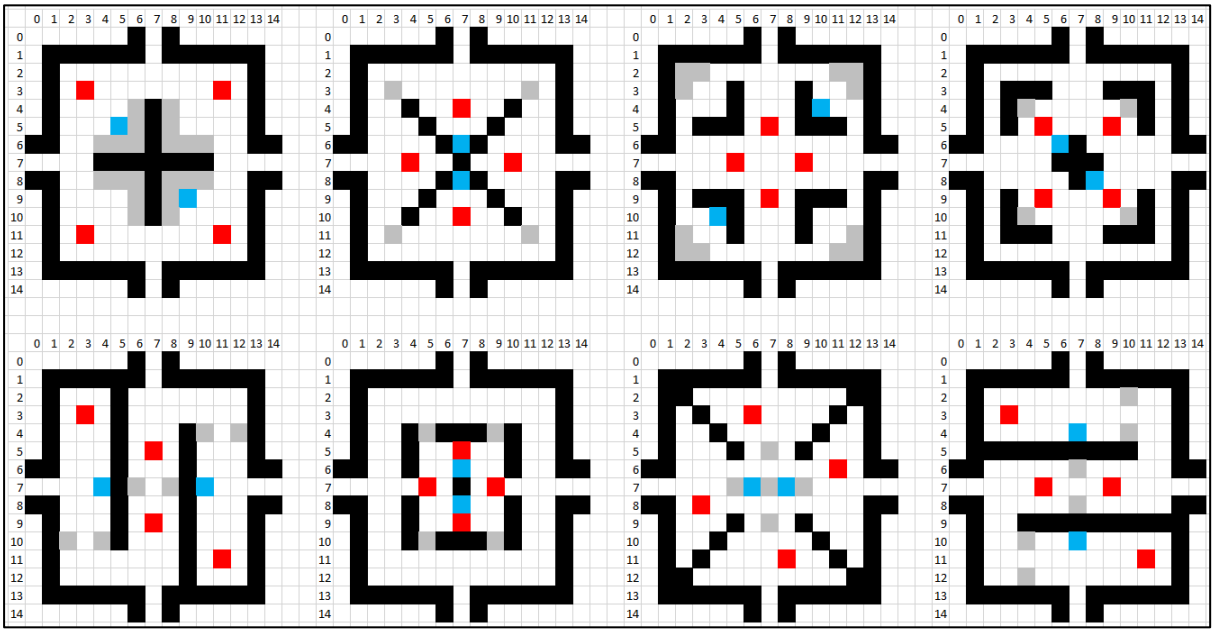
Collectables and Traps

Each room also needs to have procedurally generated collectable items and traps. The collectable items needed would be; a half heart to increase health by 1, a full heart to increase health by 2, and an arrow to increase the arrow count by 1. The trap would be a tile representing a spiked floor, which if the player walks over damages the player for 1 health. The sprites for these objects are;



Collectable sprites and spiked floor tile

It made sense for each room structure to have certain positions which could contain a trap, or a collectable. When a room is being generated each potential trap and collectable position has a chance of generating a collectable or trap. To work out these potential positions I used the Excel room template to produce a mock up;



The 8 room structures with potential spawn locations

All of the grey spaces represents potential spike floor traps, the blue spaces represent potential collectables and the red spaces represent where enemies will spawn.

Considering the traps first; in the top left example room, the spikes are gathered into 4 groups, a random number is then generated for each group between (and including) 0 and 2. If this number is 0 then that group of spikes is added to the room, if the number is anything else that group of spikes is omitted. This means that each group of spikes has a 1 in 3 chance of being spawned, so on average any room with a lateral cross will have between 1 and 2 groups of spikes. This allows for rooms with the same internal wall structure to still have some differences from one another.

Different room structures have slightly different odds of spawning a group of spikes depending on how many spike tiles make up one group and also the total number of spike groups

the structure contains. Overall though the average amount of spike groups per room should always be between 1 and 2.

The spiked floor tile also needs to be added to the player tile collision code, so that when the player makes contact with the spiked floor, 1 health can be deducted and the player can be knocked back slightly. This is done quite easily by adding this segment of code to each of the directions for player movement;

```
else if (player.damaged == false && (map[player.tileX][player.tileY] == 19 || map[(player.xpos + player.width + 12) / TS][player.tileY] == 19)) {
    player.damaged = true; player.health--; player.ypos += 3 * player.walkSpeed; damageTime1 = GetTickCount(); }
```

Another issue that this segment of code fixes, is that because the collisions are checked every render frame, if the player collides with the spiked floor they are killed within 6 frames as their health is constantly decremented. This makes it almost unplayable for the user and was not the intended outcome. Instead a "cooldown" on player damage was created. The cooldown starts a timer when the player is first damaged and changes the new player Boolean variable *"damaged"* to true. The timer waits until a second has passed and then returns "*damaged"* back to false. By limiting the player to only being able to receive incoming damage when *"damaged"* is false, this stops the spikes repeatedly damaging the player each frame. The timing loop for the cooldown works in the same way as the animation timer;

```
if (player.damaged == true) {
    damageTime2 = GetTickCount();
    if ((damageTime2 - damageTime1) > 1000)//wait 1 second
    {
        damageTime1 = damageTime2;
        player.damaged = false;
    }
}
```

The positioning of the collectables in each room is handled in the same way as the spikes. There are two potential collectable spots in each room, and each spot has a 1 in 4 chance of holding a collectable, which could be any of the three items discussed. This does however mean that a struct is needed to define a collectable;

```
struct PickupList {
    int active;
    int type; //0 = half heart, 1 = full heart, 2 == arrow
    int xpos, ypos;
};
```

The active state and world position are stored as is the pickup *"type"*. The type represents if the pick-up is a half heart, full heart or arrow, and is used to decided which sprite to draw and also what happens when a player collects it.

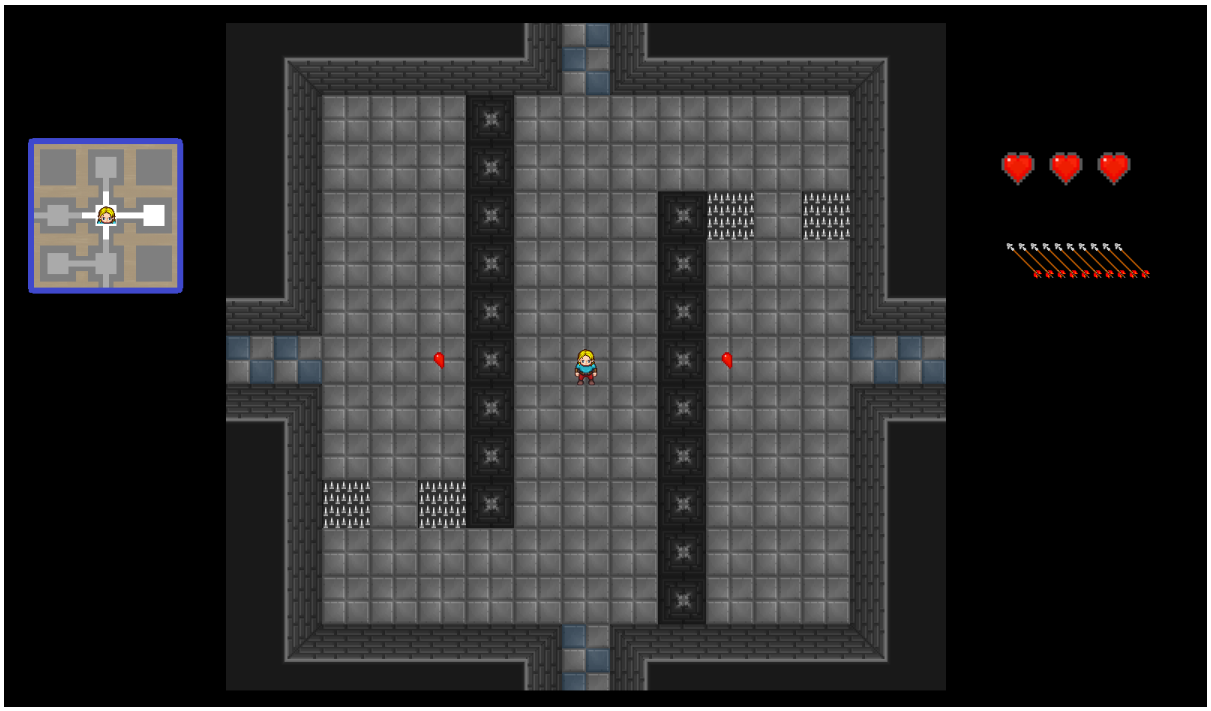
Instead of having a giant global array of all the potential pickups, an array of two *"PickupLists"* is declared within the struct for a room. This matches the design of having up to two collectables in every standard room. The only downside to this structure is that in order to reference any of the pickup variables in the player's current room now requires quite a long piece of code;

```
floorGrid[player.roomX][player.roomY].pickupList[i].type == 0
```

Player collisions with the collectables are handled in the same way as the keys; if they player enters the same tile as the collectable a collision occurs. At this point the collectable type is considered, and depending on the value the player either has their health incremented by 1 or 2, or their arrow count incremented by 1. A check is also put in place to stop the player collecting an item if it would take their health or arrow count above the respective maximum value.

```
if (floorGrid[player.roomX][player.roomY].pickupList[i].type == 0 && player.health < player.healthMax) {
    floorGrid[player.roomX][player.roomY].pickupList[i].active = 0;
    floorGrid[player.roomX][player.roomY].pickupList[i].xpos = 0; floorGrid[player.roomX][player.roomY].pickupList[i].ypos = 0;
    player.health++;
}
if (floorGrid[player.roomX][player.roomY].pickupList[i].type == 1 && player.health < player.healthMax - 1) {
    floorGrid[player.roomX][player.roomY].pickupList[i].active = 0;
    floorGrid[player.roomX][player.roomY].pickupList[i].xpos = 0; floorGrid[player.roomX][player.roomY].pickupList[i].ypos = 0;
    player.health += 2;
}
if (floorGrid[player.roomX][player.roomY].pickupList[i].type == 2 && player.arrowCount < player.arrowMax) {
    floorGrid[player.roomX][player.roomY].pickupList[i].active = 0;
    floorGrid[player.roomX][player.roomY].pickupList[i].xpos = 0; floorGrid[player.roomX][player.roomY].pickupList[i].ypos = 0;
    player.arrowCount++;
}
```

With the UI, collectables and traps implemented, the game screen showing a newly procedurally generated room looks like;

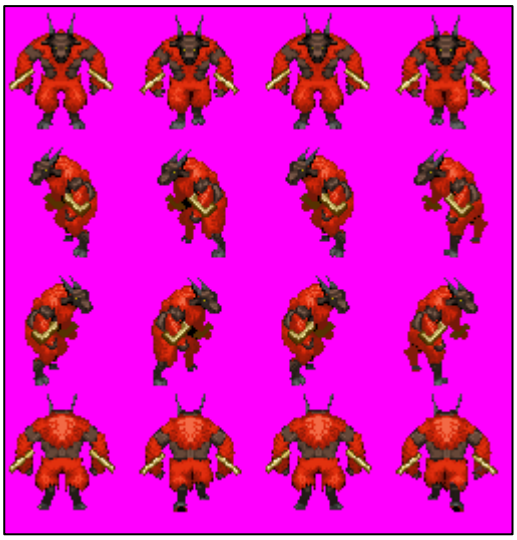


Game screen featuring UI and procedurally generated room elements

Enemy Class, AI and Collisions

I decided that a good enemy would be a minotaur, which is a quite a traditional monster for a dungeon. It would have two states, patrolling and charging. The default state would be patrolling which would involve choosing a direction, then choosing an amount to move in that direction before pausing for a random time and repeating. The charging state would be triggered if the player crosses the path any distance in front of the minotaur. In this state the enemy would move towards the position the player was located when the charge was triggered. The movement would be at a faster speed than when patrolling, and would only stop charging once the enemy collides with a wall. If the enemy collides with the player at any time, the player should have 1 decremented from their health.

The sprites I found for the minotaur have four frames per animation and feature the creature walking in all four directions;



Minotaur Sprites

Due to the complexity of an enemy it made the most sense to create a class to handle the necessary variables and functions. The variables for the enemy class are;

```
private:
    int active; // 0 inactive, 1 active
    int xpos, ypos, direction; //direction 0 - 3 (NWSE)
    int tileX, tileY;
    int destinationX, destinationY;
    int health;
    int behaviourState; // 0 NULL state, 1 patroling, 2 pursuing
public:
    static const int walkSpeed = 2, chargeSpeed = 5;
    static const int height = 50, width = 52;
    bool chosenDirection;
    int t1, t2, delay;
    int animT1, animT2, animDelay;
    int animationState; // 0 - 3 standing (NWSE), 4 - 7 walking (NWSE)
    int f, maxf;
```

The private variables are all related to the enemies positioning, health or AI, so it made sense to keep these private in order to protect them from being altered as they all have an effect on gameplay. In contrast, all the public variables are either constants or used to handle the enemy animation, which has no overall effect on gameplay.

The enemy *"health"* is set to 1, and a single arrow is enough to kill them, this variable was still included in case different types of enemy needed to be created. The destination variables are used to control the enemy movement. If the destination position doesn't match the enemy's current position, then the enemy will move to correct that. The final private variable *"behaviourState"* is the current AI mode of the enemy, 0 is NULL for when the enemy isn't yet active, 1 is patrolling and 2 is charging.

There are two constant values for the enemy speed; *"walkSpeed"* is the speed the enemy moves whilst patrolling, this is slightly slower than the player's walk speed. However, *"chargeSpeed"* is the speed when the minotaur is charging at the player, and this is slightly faster than the player's

walk speed. This means that the player can actively avoid the patrolling enemy, but needs to dodge the charging enemy. There are also two sets of timing elements, one for timing how long the enemy pauses before moving again and the other to handle the animation frames. The remaining public variables are the standard animation state and frame variables.

The enemy class also has a series of public functions, including getters and setters for all of the private variables. The unique functions are;

```
void Initalise(int xpos, int ypos);
void Destroy(void);

void ChooseDirection(void);
void StartPatrol(void);
void StartCharge(int playerX, int playerY);
```

The initialise function is called by the level generator and passes the x and y co-ordinates of the tile where the enemy should spawn. These spawn positions can be seen on the Excel chart of the room structures. The enemy is changed to active, their health and behaviour state is set to 1, the enemy position variables are set using the given parameters, and a random direction is selected;

```
void Enemy::Initalise(int xpos, int ypos)
{
    active = 1;
    health = 1;
    this->xpos = xpos;
    this->ypos = ypos;
    tileX = (xpos + 20) / TS;
    tileY = (ypos + 30) / TS;
    direction = rand() % 4;
    destinationX = xpos;
    destinationY = ypos;
    behaviourState = 1;
}
```

The destroy function is called when an enemy is killed. The enemy is set to inactive and the behaviour station and position are set to 0.

Choose Direction is used to select a new direction for the enemy when they are patrolling and have reached their current destination. A new random direction is chosen, but if it is the same as the previous direction it is randomised again until a new direction is found. This helps prevent the enemy repeatedly walking into walls. The animation state is then also changed to match the animation for the new direction;

```
void Enemy::ChooseDirection(void)
{
    if (behaviourState == 1) {

        int directionNew = rand() % 4;
        while (direction == directionNew) { directionNew = rand() % 4; }
        direction = directionNew;

        f = 0;
        animationState = direction;
    }
}
```

Start Patrol chooses a random distance for a patrolling enemy to move, between 1 and 5 tiles. Depending on the direction the enemy is facing, either the destination x or destination y variables are changed to make the enemy move in that direction. For example, if the enemy was facing west, the destination x value needs to be set to the movement distance less than the enemies current x position. The animation state is also changed to the direction appropriate walking animation;

```
void Enemy::StartPatrol(void)
{
    if (behaviourState == 1) {

        if (direction == 0) {
            destinationY = ypos - ((rand() % 5 + 1) * TS);
        }
        if (direction == 1) {
            destinationX = xpos - ((rand() % 5 + 1) * TS);
        }
        if (direction == 2) {
            destinationY = ypos + ((rand() % 5 + 1) * TS);
        }
        if (direction == 3) {
            destinationX = xpos + ((rand() % 5 + 1) * TS);
        }
    }
    animationState = direction + 4;
}
```

The Start Charge function is called whenever the player walks in front of the enemy and triggers a charge. This function is passed two parameters; the players current x and y position. Depending on the enemy's current direction, the direction variables are changed to the player's position plus 12 tiles. This guarantees that the enemy will charge through the player's position and keep going until they make contact with a wall. The enemy behaviour state can then be set to 2 because it is now in charge mode, and the direction appropriate walking animation can be selected;

```
void Enemy::StartCharge(int playerX, int playerY)
{
    if (direction == 0) {
        destinationY = playerY - (TS * 12);
    }
    if (direction == 1) {
        destinationX = playerX - (TS * 12);
    }
    if (direction == 2) {
        destinationY = playerY + (TS * 12);
    }
    if (direction == 3) {
        destinationX = playerX + (TS * 12);
    }
    SetState(2);
    animationState = direction + 4;
}
```

Each room can have up to 4 enemies, the amount is decided when the room is being generated using the same technique as when choosing collectables. This means that the struct for a room also needs an array of 4 enemies.

One downside to using this structure was that the code was now becoming unwieldy. For example, I only wanted the enemy movement and collisions to be calculated for the room the player is currently in, so referencing an enemy required the following code nested in a loop to check each enemy;

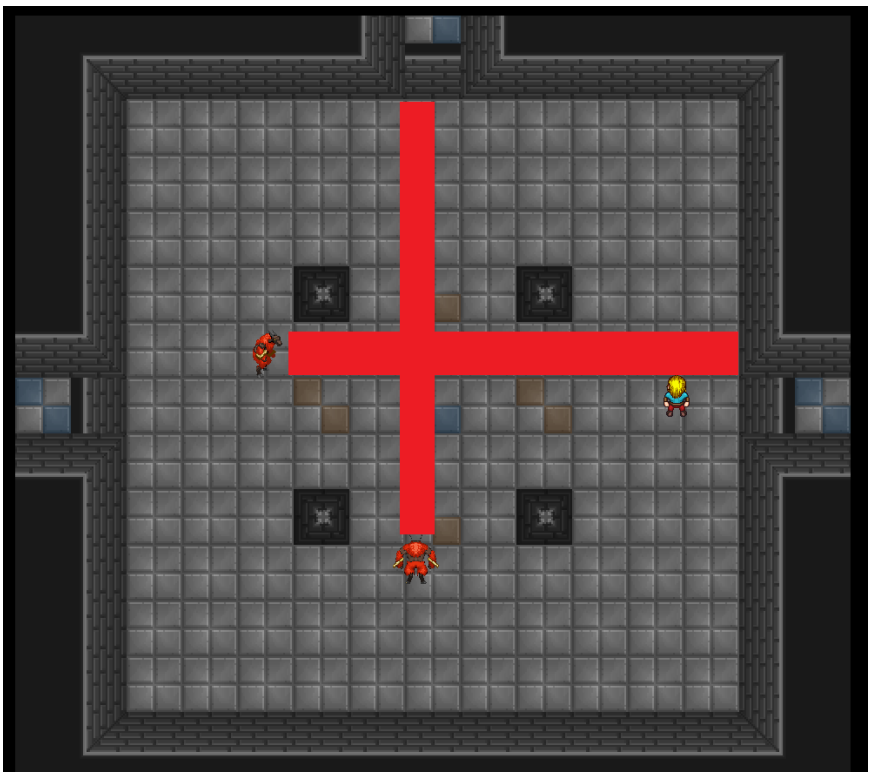
```
floorGrid[p.roomX][p.roomY].enemy[i]
```

In order to try to make the code slightly shorter I renamed the instance of the player struct from “player” to “p”. This helped, however the code remained quite long and difficult to read.

All the enemy movement and collision is contained within a loop which checks all 4 of the possible enemy positions in each room. Each position is first checked to see if the enemy is active. Then, if the enemy has already finished their current movement (the enemy destination co-ordinates match the enemy position), then a new direction is selected using *“ChooseDirection”* and the enemy behaviour state is set to patrolling. This makes sure the enemy returns to patrol mode whenever they finish moving from either a patrol or a charge.

The timing element for movement is constantly cycling every second. This determines how long an enemy has to wait between choosing a direction and starting movement. This means that the enemies will pause for any time up to a second. Once the pause has finished the *“StartPatrol”* function is called to begin the enemy’s next movement.

When the player enters the space in front of the enemy the charge needs to be triggered;



Example of enemy field of view

If the player enters the red areas in the image above, then the enemy will begin charging. The definition for each of these areas is different depending on the direction the enemy is facing, but they all follow the same principle. Consider an enemy walking east;

$$if\ (((player\ y\ >\ enemy\ y\ \&\&$$

$$player\ y\ <\ enemy\ y\ +enemy\ height)\ ||$$

$$(player\ y\ +\ player\ height\ >\ enemy\ y\ \&\&$$

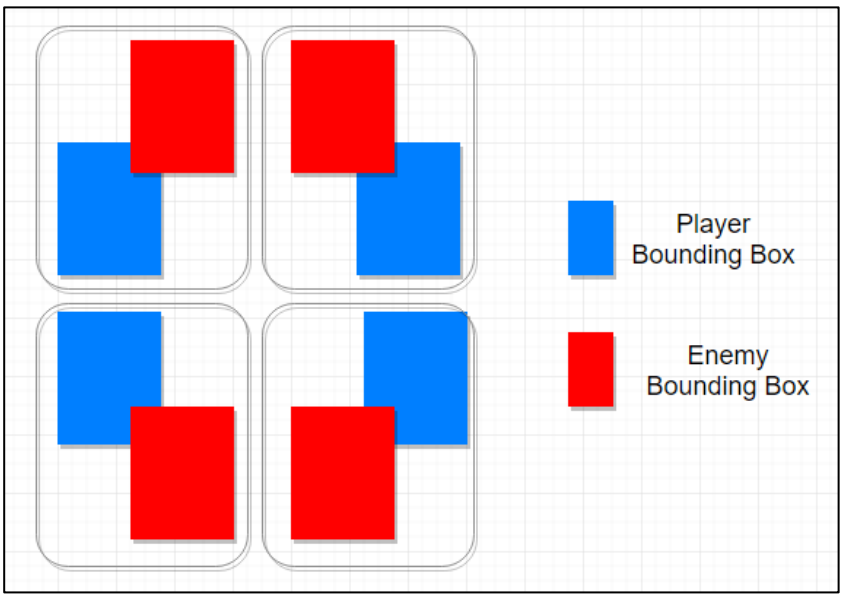
$$player\ y +\ player\ height\ <\ enemy\ y\ +\ enemy\ height))\ \&\&$$

$$player\ x >\ enemy\ x)$$

The first four lines of the condition cover the definition of the top and bottom of the enemy's field of view relative to the player's top and bottom co-ordinates. The x co-ordinate check is used to make sure the player is in front of the enemy and not behind them. If this condition is met, then the enemy's behaviour state is set to charge and is passed the players current co-ordinates.

As mentioned earlier, the movement is performed by comparing the enemies current position against the destination position. If it doesn't match, then the enemy moves towards the destination. Depending on the enemy's behaviour state they either move at *"walkSpeed"* or *"chargeSpeed"*.

The enemy wall collisions are handled using the tile collision method already implemented for the player and arrows. However, colliding with the players is achieved using bounding box collision. This considers the four possibilities of when the player is colliding with an enemy;



Four possible bounding box collisions

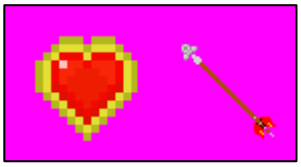
Consider the top left scenario. This collision is occurring when the player's x position plus the player width is between the enemy's x position and the enemy's x position plus the enemy width. It also has to be when the player's y position is between the enemy's y position and the enemy's y position plus the enemy height. The code for this example appears as;

```
if(p.xpos + p.width > floorGrid[p.roomX][p.roomY].enemy[i].GetXPos() && //top right corner of player collision
    p.xpos + p.width < floorGrid[p.roomX][p.roomY].enemy[i].GetXPos() + floorGrid[p.roomX][p.roomY].enemy[i].width &&
    p.ypos > floorGrid[p.roomX][p.roomY].enemy[i].GetYPos() &&
    p.ypos < floorGrid[p.roomX][p.roomY].enemy[i].GetYPos() + floorGrid[p.roomX][p.roomY].enemy[i].height){

        p.damaged = true; p.health--; damageTime1 = GetTickCount(); break;
}
```

By changing the player's position values to check different corners of the bounding box, all four possible collision scenarios can be covered.

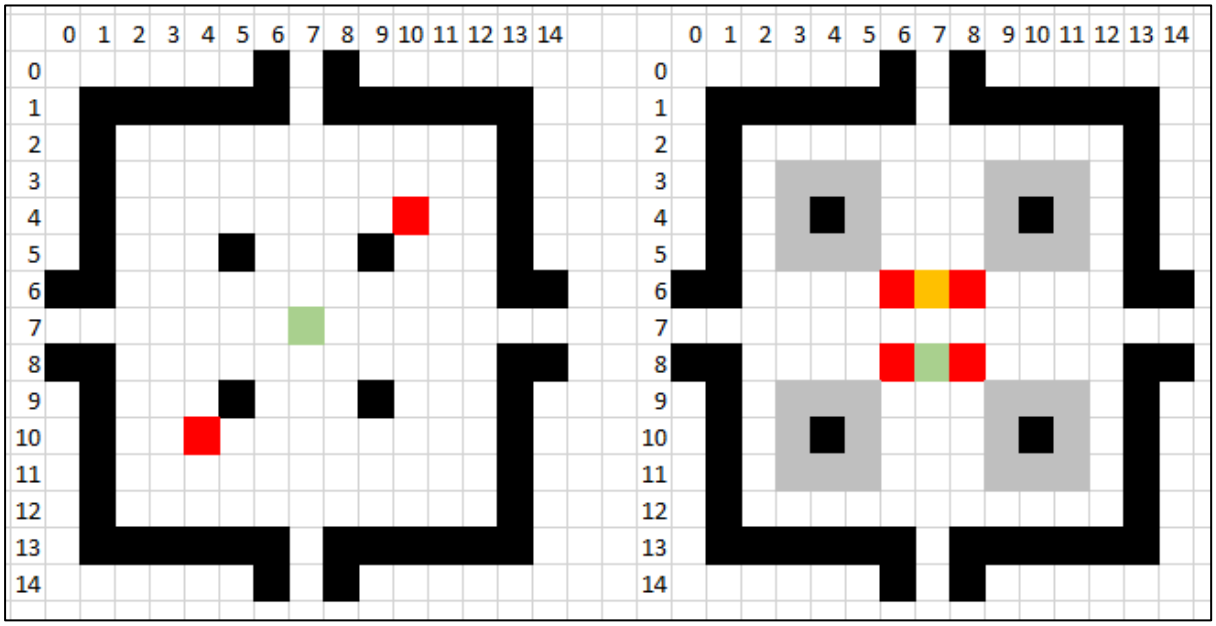
Final Touches

To add some player progression to the game two new collectables were added, one which increases the player's max health by 2, and one which increases the player's max arrow count by 5;



Max Health and Max Arrow Increase Collectables

Some changes were also made to the key rooms. Two possible structures were created; one which always contains two enemies and one which contains four enemies and a spiked floor. The more difficult of these rooms also spawns one of the above collectables as well as the key. Whenever a key room is designated by the level generator, there is a 1 in 4 chance that it could be the more difficult room. The structure of the keys rooms is;



Key room structure

The green space represents where the key will spawn and the orange space is where the collectable will spawn. As with the other rooms, the enemies need to be killed first in order for the doors to open or for the key and collectable to be spawned.

The game over scenario was also implemented. If the player's health reaches 0, then a new variable called *"gameOver"* is set to true, and the player death animation is played. Whilst *"gameOver"* is true, no movement or player input is calculated, and pressing the space bar will restart the game generating a new first floor.

Finally, two text displays were added. The first is called when *"gameOver"* is true and the second is called every time a new level is reached;

"Game Over" and "Next Level" text bitmap

These were drawn to the screen using a slightly different DirectX function called *"Blt"*, which allows for a bitmap to be resized in game. This function works in a similar way to *"BltFast"*, but instead of just providing two screen co-ordinates as the destination, a "RECT" is used to define an area on the screen. The source bitmap is then stretched or shrunk to fit into the defined area. Using this feature the text can be drawn to the screen with some animation.

The game over text grows up to a maximum limit, then shrinks down to a minimum limit. This achieved by incrementing a variable *"size"* by a small constant amount each frame, and adding this value to the destination "RECT";

```
if (gameOver == 1) {

    r.left = 272;
    r.top = 787;
    r.right = 780;
    r.bottom = 890;

    size += (0.13 * sizeDirection);

    if (size > 25.0 && sizeDirection > 0) sizeDirection = -1;
    if (size < 0.3 && sizeDirection < 0) sizeDirection = 1;

    rTextSize.left = (SW/2) - 254 - size - (TS * 2.5);
    rTextSize.top = (SH/2) - 52 - size;
    rTextSize.right = ((SW/2) + 254) + size - (TS * 2.5);
    rTextSize.bottom = ((SH/2) + 52) + size;
    fx.dwSize = sizeof(fx);

    DDB->Blt(&rTextSize,DD1, &r, DDBLT_KEYSRC, &fx);
}
```

The next level text starts small and grows until a maximum size limit is reached, then the text disappears. This is achieved in the same way as the game over text, but without reversing the direction of *"size"* growth.

When the player has died the screen now appears as;

Game Over Screen

One last UI element was also added to show how many keys the player had left to collect in order to open the exit. This was shown using padlocks under the mini-map, with each padlock representing one key that the player still has to find. This was drawn to the screen exactly the same way as the arrow UI.

The source files for this game are “main.cpp”, “levelgenerator.h”, “levelgenerator.cpp”, “enemyclass.h” and “enemyclass.cpp” and are attached to this file along with the complied application “DDEX4” and the source bitmap “All”. This application will only be displayed as intended on a monitor with a resolution of 1920 x 1080.

Evaluation

Overall this game functions as required. It has repeatable gameplay thanks to the procedural generation of the levels and contains many of features of classic 8-bit games. There are however some errors which could be corrected and also some improvements which could be made to gameplay.

In terms of gameplay; every room is currently very similar, mostly due to the single enemy type. It would add more challenge if different enemies were added which used a different form of AI. The implementation wouldn’t be too difficult as a new enemy could use class inheritance to be derived from the existing enemy class.

The difficulty could also increase more as the player reaches deep floors. At the moment the only increase in difficulty is the amount of floors the player has to navigate in order to find the keys and the exit. To reconcile this; enemies could be made more likely to spawn as the player reaches deeper floors.

To add some more variety to the game, different room structures could be used for different floors. For example, the first 2 floors could be the standard dungeon, then the next 2 floors could

use different internal wall structures, spawn different enemies and even use a different tile set. This could be used to create different groups of floors such as dungeon, jail, cave etc. The order which these floors appear in could also be decided procedurally.

Some game mechanics could also be improved. Currently the tile engine has the problem of not being able to fit a whole number of tiles into the screen space. The work around being used is to draw black bars over the edges of the screen to cover this offset. A better solution would be to dynamically select part of a tile to fill this gap. This wouldn't be too difficult to implement as the offset size in pixels is constantly being updated and stored in *"sxmod"* and *"symod"*. Conditions would need to be made for placing different edge tiles. These edge tiles could then have their "RECT" defined taking into account this offset, so that a partial tile is used to exactly fill the gap. For example, the "RECT" definition for an edge tile on the right of the screen would be;

```
r.left = TS*z;
r.top = 0;
r.right = (TS*z + TS) - sxmod;
r.bottom = TS;
```

This would allow for the screen to be perfectly tiled from edge to edge if this process was applied correctly to all the edge tiles.

The enemy collision with the player is also not fantastically accurate. This is because the bounding boxes defined for the player and enemy do not perfectly match their actual shape. An alternative collision method could be used such as masking. A black shape would need to be created for each frame of the enemy and player's animations. These shapes known as "masks" would then line up and move with the enemy and player respectively. Each frame both of these mask positions are fed into a Boolean "AND" gate. If any black pixel of the two masks overlap, then the "AND" gate returns true and a collision is detected. This process can allow for pixel perfect collision.

Another improvement would be the ability to dynamically print text to the screen. For example, it would be useful to display the players current floor as a number printed to the screen. This would require creating bitmaps for the numbers 0 to 9. They would all have to been the same size to make sure they lined up when printed. Printing between 0 and 9 would be easy to implement as the position of the sprite would related directly to the number being print. For values of 10 or higher, a more complex logic would need to be used to define the ten value before printing the unit value.

References

Harrison, M. (2003) *Introduction to 3D Game Engine Design using DirectX 9 and C#.* Kinetic Publishing Services.

Luna, F. (2008) *Introduction to 3D game programming with DirectX 10*. Jones and Bartlett Publishers.

Snook, G. (2003) *Real-Time 3D Terrain Engines Using C++ and DirectX 9.* Charles River Media Inc.

http://gaurav.munjal.us/Universal-LPC-Spritesheet-Character-Generator/ (accessed on 25/11/16)

http://www.directxtutorial.com/Lesson.aspx?lessonid=9-4-8 (accessed on 12/12/16)

https://msdn.microsoft.com/en-us/library/windows/desktop/bb147399(v=vs.85).aspx (accessed on 18/12/16)

Appendices

2D Room Structures Spreadsheet – *"Room Grid.xlsx"*

2D Platformer Source Code Files - *"main.cpp", "levelgenerator.h", "levelgenerator.cpp", "enemyclass.h" and "enemyclass.cpp"*

2D Platformer Bitmap – *"All.bmp"*

2D Platformer Compiled Application – *"DDEX4"*