COMP3600 Milestone 2

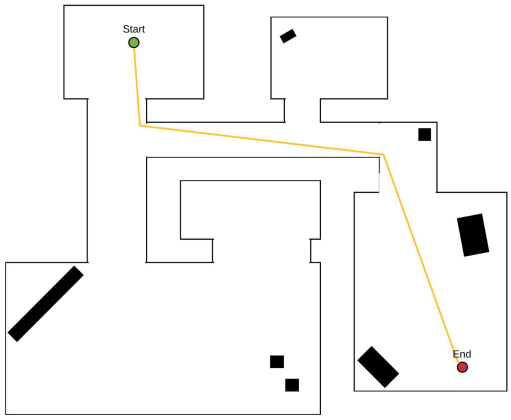
Jack Kilrain (u6940136)

Overview

This project will be built on top of a ray casting implementation, with the intention of extending it beyond the raw single colour, basic rectangular polygon fills. Texturing will be added to allow for more comprehensive rendering akin to that of the id tech 1 engine that DOOM is based on. Additionally, path solving will be implemented with the A* path finding algorithm in order to find the most efficient path from a starting point to an end point in a level.

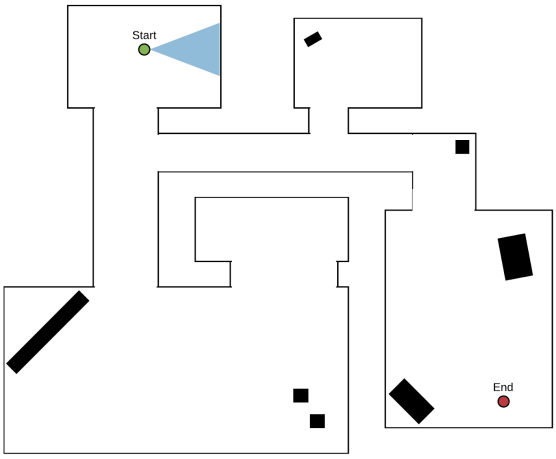
This project will utilise C++ as the language of choice, since a lot of low-level optimisation is required in order to produce an effective algorithm. Additionally, the use of 8GB of RAM is desired in order to ensure memory capacity has sufficient working space.

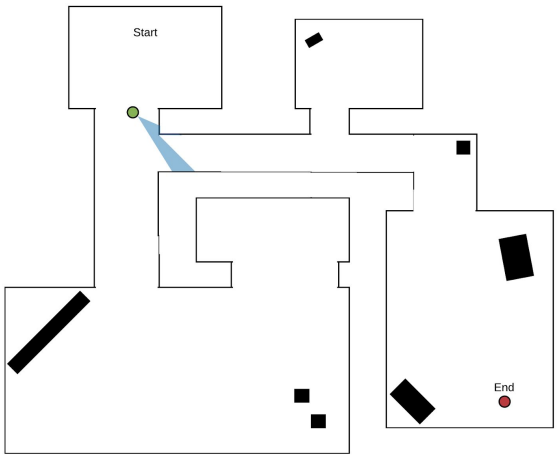
Usage with example level

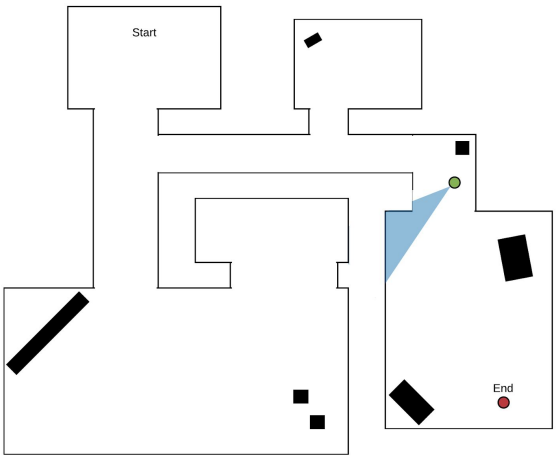


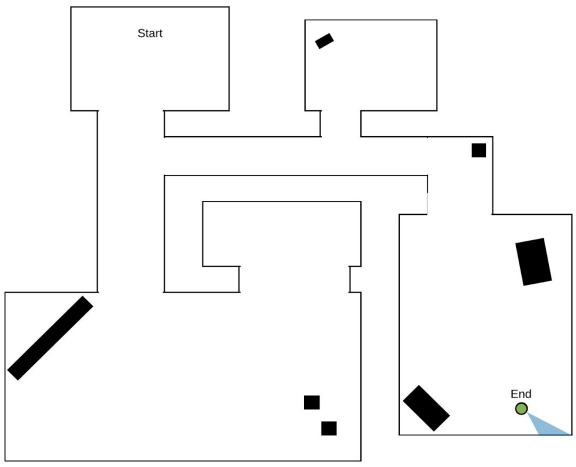
Utilising these elements, the idea is to create an algorithm that finds the minimum amount of polygons required to be rendered in order to go from the starting point to the finish point. In order to illustrate this point, let us consider the set of FOV overlays on a level design below.

Here A* is used to determine the shortest path, with an additional heuristic applied to the continuously generated BSP tree for the polygons that need be rendered as the player progresses. Using the A* algorithm on this level design produces a path similar to the following (note the use of low-fidelity point-to-point deviation for the path). An example path (roughly the minimal rendering required is shown below on the maps, this is in order left to right, top to bottom).









Functionality

Hashing

As the raycaster is handling all rendering functionality, this includes loading, referencing and retrieving textures. In order to handle the textures efficiently, rather than storing a full instance of them on each wall instance, a reference is stored in the form of the texture name. This is then used to retrieve the texture instance via a hashtable matching a low length string to a texture instance. In this instance, we will be using a string of maximum length 32, allowing for amortized O(1) hashing as per the usual implementation of a hashmap/hashtable. It's important to note that only standard non-extended ASCII values as described by the default char data type in C/C++ on an english locale.

In order to reduce collisions with low-sized tables, a modified version of rolling polynomial hashing is used. The standard implementation of rolling polynomial uses continuous offsets as a factor as a prime power. This is extended by offsetting the current key by 60 (integer value of 'a' - 1), increasing the modulo rate when multiplied by a prime power.

Below is a standard implementation of the rolling polynomial method, and following it a modified version using methodology from the FNV-1a algorithm

```
int rollingPolynomial(string key) {
    int p = 31;
    int m = 1e9 + 9;
    int prime_power = 1;
    int hashVal = 0;
    for (each character index in the key as i) {
        hashVal = (hashVal + (key[i] - 'a' + 1) * prime_power) MOD m;
        prime_power = (prime_power * p) MOD m;
```

```
    }
    return hashVal;
}
```

In order to make this more concrete for our purposes, we will rely on pre-evaluated prime values and offsets used with the FNV-1a hashing algorithm.

```
constant int FNV_PRIME_MOD = 22801763489;
constant int FNV_PRIME_32 = 16777619;
constant int FNV_OFFSET_32 = 2166136261;

int rollingPolyFNV1a(string key) {
    if (length of key > 32) {
        Return -1;
    }
    int prime_power = 13;
    int hashVal = FNV_OFFSET_32;
    for (each character index in the key as i) {
        hashVal = (hashVal + (key[i] XOR FNV_PRIME_32) * prime_power) MOD table_size;
        prime_power = (prime_power * FNV_PRIME_MOD) MOD table_size;
    }
    return hashVal;
}
```

This gives a solid basis to build a modified XOR based hashing method on top of, allowing for a more diverse set of integers to be achieved. Note that this hashing relies on the irregular occurrence of XOR primes from the base usage of a standard prime (16777619 in this case). This arises as from XOR multiplication not using carry between bits, which generally does not produce regular primes since base 2 multiplication has indirect prime offsets when represented on a bit-by-bit basis.

A* Shortest Path

A large part of the project is using the A* algorithm and extending the base heuristic to account for polygons in the range of the player frustum. In order to do this, a base implementation of the A* pathfinding algorithm needs to be used.

The A* algorithm modifies Dijkstra's pathfinding to use not only forward (current location relative to end target) heuristic but also the reverse relative heuristic from current location to starting location. This simple idea is expressed through the equation $F = G + H$, where G is the path distance between the current position and the starting position, H is the estimated distance between the current position and the end position and F is the total cost, the sum of the two. This simple formula allows for relative evaluation at every note making choosing the next position to try much easier since the F cost scales almost inversely when moving away from the target.

Using the A* algorithm in this project is a fairly straightforward choice, notably because it works well with graph style data. In this case the graph is grid aligned fixed width and height squares. Something to note is that this uses conditional neighbouring and not Manhattan distance 1 neighbours. This is the case as it allows for diagonal movement and the conditional part allows for evaluation of relative wall position in the squares around, preventing movement between two diagonally connected walls. Figure 1 provides an example of this conditional invalidation for diagonal movement compared to normal movement availability.

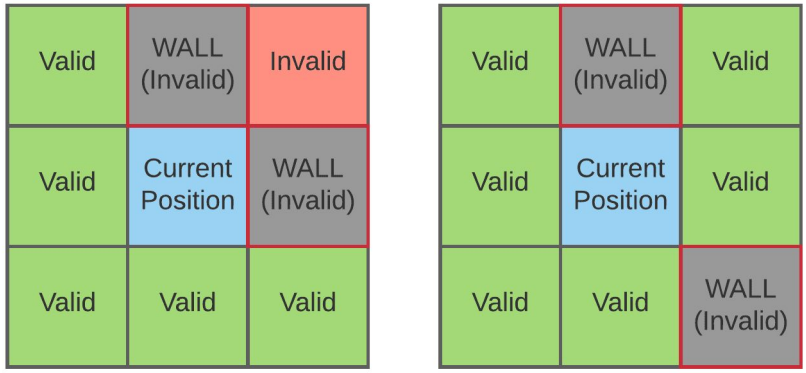


Fig (1): Examples of neighbour configurations with diagonal invalidation

As discussed above, the standard heuristic considers only distance between start, current and end nodes. However, in this project there is an additional heuristic parameter to consider, that is the difference in polygon count while traversing to this particular position. In order to evaluate this additional parameter, each step of the A* algorithm will traverse the BSP tree relative to the current position, generating a convex hull of all polygon vertices (and subsequently polygons) within the frustum. This can be added to the running count of the polygons being rendered as a set union, ensuring no duplicate polygons are counted. Continuous evaluation using this methodology will allow for pathing to be relative to render overhead.

Something important to note for the implementation, as of this milestone, the BSP tree has not been implemented and as such the heuristic for the polygon count has not been implemented. However, the standard A* algorithm has been and works accordingly.