COMP3600 Final Project - Milestone 1

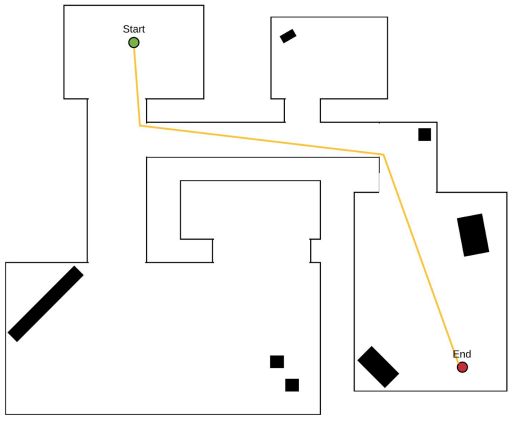
Jack Kilrain (u6940136)

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.

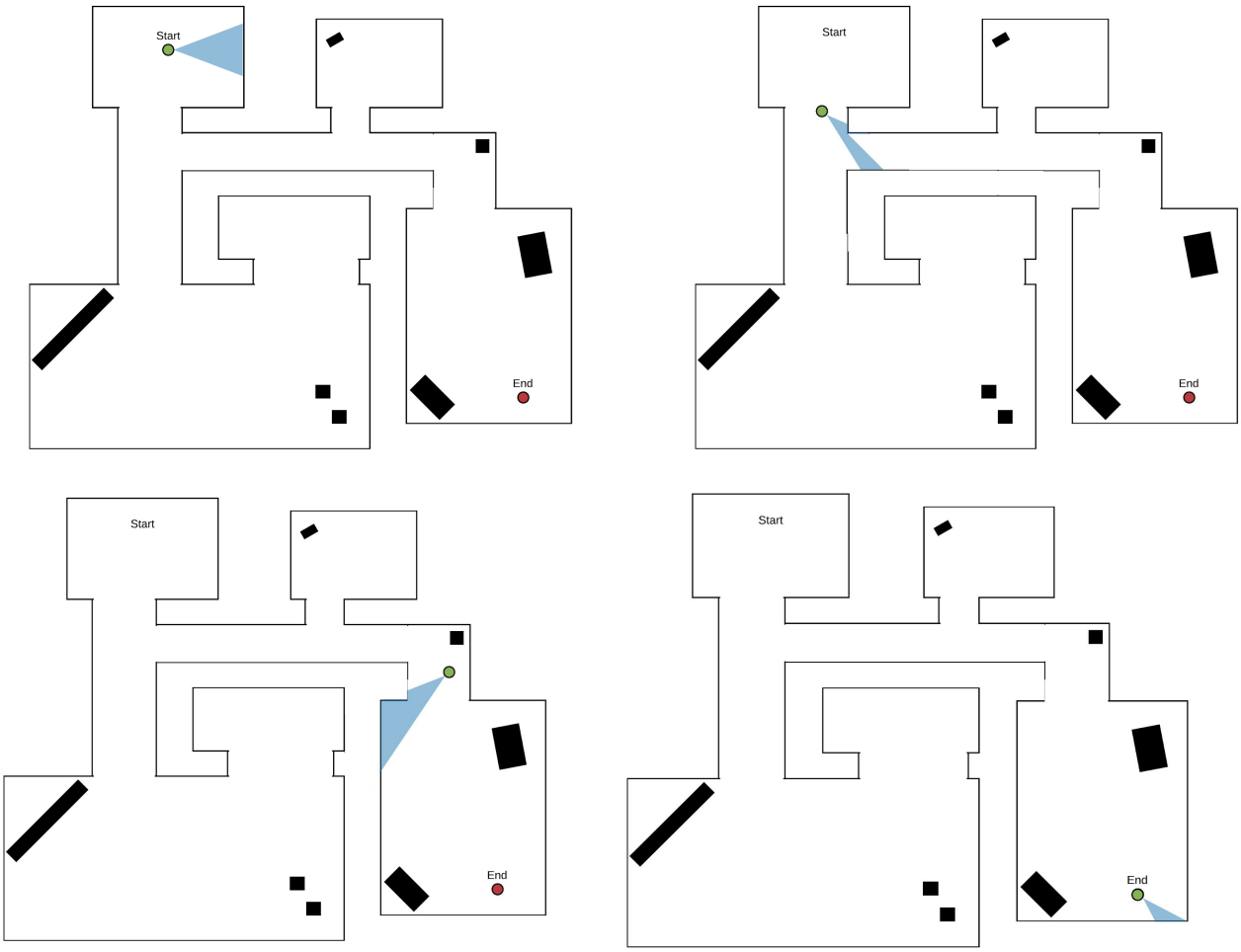
Usage with example level

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).

Now let us consider the FOV of the player and the BSP tree that is generated from the main points of the path. Here the calculation is limited to differential geometry, only recalculating the convexity of the polygons (to find the minimal amount of convex sub-polygons to render) when a particular polygon ID does not appear in the current list of minimal polygons. An example of this is in the second image (order is top to bottom, left to right) where the player views new polygons in the hallway (subsequently having new IDs/hashes) which cause a complete rebuild of the BSP tree rather than a subtree rebuild.



Continually applying this process, the approximate minimal render count of polygons can be found in order to traverse the level.

Functionalities

BSP Tree

BSP trees are used to find minimal sub-polygons of the render space to render, minimising the amount of arbitrary polygons to render at once. BSP trees

Hashing

As the algorithm progresses, there are continual updates to the polygon list. Normally these polygons contain total information, regarding the coordinates to construct the outside and texture reference.

In order to reduce the total amount of data duplication and as such the memory and time overhead, a hashing algorithm will be used to generate unique “ID’s” for the polygons. This not only makes it easier to create a heuristic for, but also means identifying overlaps (existing polygons in the list) is easier and more efficient.

Let us consider an example polygon object (C/C++ style implementation for simplicity):

```
#include <utility>
#include <string>
#include <cstring>
#include <vector>

using namespace std;
using coords = pair<float, float>;

class Polygon {
   public:
       Polygon(const string& tex_ref, vector<coords> hull_points);
       ~Polygon();
       size_t hashPolygon();
   private:
       string convertHullPoints();
       size_t hashString(string string_to_hash);
       string _tex_ref;
       vector<coords> _hull_points;
};

Polygon::Polygon(const string& tex_ref, vector<coords> hull_points) {
   this->_tex_ref = tex_ref;
   this->_hull_points = hull_points;
}

Polygon::~Polygon() {
   delete &_tex_ref;
   delete &_hull_points;
}

string Polygon::convertHullPoints() {
   string return_val = "";
```

```
    for (coords point : this->_hull_points) {
        return_val.append(to_string(point.first));
        return_val.append(to_string(point.second));
    }
}

size_t Polygon::hashString(string string_to_hash) { // <-- Method to implement
    size_t hashed_string;
    // ... Do hash on string ...
    // ... Hashing implementation goes here ...
    return hashed_string;
}

size_t Polygon::hashPolygon() {
    string total_polygon_string = this->_tex_ref;
    total_polygon_string.append(convertHullPoints());
    return hashString(total_polygon_string);
}

Polygon example_polygon(
    "example_texture",
    vector<coords>{
        coords(2.1, -0.32),
        coords(0.36, 3.74),
        coords(-4.01, 5.21)
    }
);
```

Hashing a polygon will utilise the following method:

1. Create new string copy of tex_ref as the base for the string to hash (let it be called returnVal)
2. Convert the hull_points to a string and append to returnVal in the format: `"<tex_ref><conv_hull_points>"`
    a. Example of converted hull points:
        i. hull_points = `[(2.1,-0.32),(0.36,3.74),(-4.01,5.21)]`
        ii. conv_hull_points = `"2.1-0.320.363.74-4.015.21"`
    b. Example of returnVal:
        i. tex_ref = “example texture”
        ii. conv_hull_points = `"2.1-0.320.363.74-4.015.21"`
        iii. returnVal = `"example texture2.1-0.320.363.74-4.015.21"`

3. Perform a hash on returnVal
4. Return the hash of returnVal

The method that will be implemented is the hashString() function, utilising a method such as polynomial rolling.

A* path finding

In order for the algorithm to be as efficient as possible with finding the shortest path, A*, a modified version of Dijkstra's algorithm is used. Rather than dijkstra's comparative enumeration of potential routes, A* heuristically evaluates transitions at each iteration. In the case of path finding in levels/mazes this allows for a much more comprehensive search and also guarantees the shortest path, rather than approximates it.

Utilising A* within the search will require some modification in order to have a heuristic that can evaluate the convexity of a BSP tree. As such the implementation will be dependent on an iteratively enumerated BSP tree based on player FOV. Here A* will compare polygon hashes will existing hashes and also total sub-polygon count from the convex mesh of the BSP tree.

References

BSP Trees

- https://en.wikipedia.org/wiki/Binary_space_partitioning
- https://iq.opengenus.org/binary-space-partitioning/
- https://ukdiss.com/examples/binary-space-partitioning-algorithm.php
- https://iq.opengenus.org/binary-space-partitioning/
- https://github.com/sudeshnapal12/Space-Partitioning-Algorithms/tree/master/BSP
- https://vgc.poly.edu/~csilva/papers/bsp-fill.pdf

Hashing

- https://stackoverflow.com/questions/998662/what-is-image-hashing-used-for
- https://en.wikipedia.org/wiki/Hash_function#Radix_conversion_hashing
- https://cp-algorithms.com/string/string-hashing.html
- https://en.wikipedia.org/wiki/Rolling_hash

A*

EXTRAS

Npm library to merge multiple c++ files:

- https://www.npmjs.com/package/codingame-cpp-merge