### Technical Design Document

1.0 Revision History

|  |  |
| --- | --- |
| Version | Description |
| 1.0 | Initial document |
| 2.0 | Update to be more satisfactory |

2.0 Development Environment

2.1 Game Engine

None

2.2 IDE

Visual Studio

2.3 Source Control procedures

Git and GitHub

2.4 Third Party Libraries

SDL (<https://libsdl.org>) - Stable, cross platform and has many online resources available.

stb\_image (<https://github.com/nothings/stb>) (for loading images.) - Familiar to me and easy to use.

stb\_truetype (<https://github.com/nothings/stb>) (for loading fonts for the menu and HUD) - Familiar to me and easy to use.

2.5 Other Software

16x16 Dungeon Tileset (<https://0x72.itch.io/16x16-dungeon-tileset>)

Tiled (<https://mapeditor.org/>)

Tiled binary exporter (<https://gist.github.com/veridisquot/374a246b95c53886af60248e1861dd67>)

3.0 Game Overview

3.1 Technical Goals

2D rendering

3.2 Game Objects and Logic

Guards, Player, Map, Key, Exit.

3.3 Game Flow

The player will have to find the key in order to unlock the exit, avoiding detection by the guards in the process. They will not be able to damage the guards in any way. The guards will kill the player if they get too close.

4.0 Mechanics

The player can pick up a key. When they touch the exit door while holding said key, the door will open and they will the game. Otherwise, if they are caught by the guards, they will lose.

5.0 Graphics

The game will use simple 2D rendering using SDL’s built-in renderer.

Entities:

There will be a generic entity class, which contains an SDL\_Texture\* for the entity’s image, as well as an SDL\_Rect for the rectangle where the sprite is located on that image. The player and all AI agents will inherit from this class.

There will be a class named “world” that will contain a vector of AI agents as well as an instance of the “player” class and a “level” class. This class will be responsible for updating all the entities and keeping track of them.

6.0 Artificial Intelligence

The AI will use A\* pathfinding implemented on a grid, with a dedicated tile layer of tiles where the agents can walk.

The agents will use a finite state machine with a simple switch statement in each agent. Each agent will be in one of three states at any given moment: patrol, chase or search. Agents by default are in the “patrol” state, where they pathfind to random locations on the map. When the player goes into the agent’s field of vision, the agent goes into the “chase” state, in which it will chase the player based on a rudimentary sterring bahaviour. When the agent loses sight of the player, it will pathfind to the last location of the player and transition into the “patrol” state.

7.0 Physics

Simple AABB collision detection and resolution will be sufficient for all physics needs.

8.0 Items

The player will be able to pick up the key by walking over it, and then unlock the exit and win by touching it.

|  |  |
| --- | --- |
| Item | Description |
| Key | Allows the player to unlock the door |

10.0 Levels

Levels will be created using the Tiled map editor, which allows easy placement of tiles on a grid. They will be exported using this binary exporter: <https://gist.github.com/veridisquot/374a246b95c53886af60248e1861dd67>

The level will be stored in an array of tile layers. Each tile layer will have an array of tiles in it. Each tile will be a structure that contains an ID of the tile set it comes from, and ID of the tile itself. There will also be an array of rectangles for map collisions, and an array of Booleans specifying where the AI agents can and cannot walk.

Tileset struct:

|  |  |  |
| --- | --- | --- |
| **Member** | **Description** | **Type** |
| image | Image containing all the tiles in the tileset; A texture atlas, in other words. To be rendered to the screen by SDL\_RenderCopy. | SDL\_Texture\* |
| tile\_size | The size of each tile in the tileset. | int |
| name | The name of the tileset | char\* |

Tile struct:

|  |  |  |
| --- | --- | --- |
| **Member** | **Description** | **Type** |
| tileset\_id | The ID of the tileset that this tile belongs to. | int |
| id | The tile’s ID. | int |

Tile layer struct:

|  |  |  |
| --- | --- | --- |
| **Member** | **Description** | **Type** |
| tiles | All the tiles used by the layer | tile\* |
| width | The width of the layer. | int |
| height | The height of the layer. | int |

Level class:

|  |  |  |
| --- | --- | --- |
| **Member/Method** | **Description** | **Type** |
| tilesets | All the tilesets used by the level. | std::vector<tileset> |
| layers | All the layers to be rendered | std:vector<tile\_layer> |
| collisions | The collision rectangles | Std::vector<rect> |
| draw(SDL\_Renderer\* renderer) | Render the level to the screen by iterating all the layers and drawing them. | Method, returns nothing. |
| load(const char\* filename) | Load the level from a file. | Method, returns a Boolean; True on success, false on error. |

Levels will be loaded like so:

* + Load the amount of tilesets (first four bytes)
  + Iterate for less than the tileset count
    - Load the tileset name size and the name data (four bytes + name data)
    - Load the image path size and the image path data (four bytes + image path data)
    - Load the image from the path
    - Load the tile size
    - Using the data listed above, create a new instance of the tileset class and push it back into a vector
  + Load the layer count (four bytes)
  + Iterate for less than the layer count
    - Load the layer name size and the layer name (four bytes + layer name data)
    - Load the layer type id. A value of zero means it’s a tile layer, a value of one means it’s an object layer
    - If the layer name is equal to "collisions” and the layer is an object layer
      * Load the object count (four bytes)
      * Iterate for less than the object count
        + Load the object x, y, width and height into a new rectangle and push it back into the collisions vector
    - Else, If the layer name is equal to “enemies” and the layer is an object layer
      * Load the x and y coordinates, and spawn a new agent at that location.
    - Else, if the layer is a tile layer
      * Read the layer width and height (4 bytes + 4 bytes)
      * Create a new instance of the tile layer struct
      * Allocate the tile layer struct’s tile buffer using the width and height.
      * Iterate for less than the height
        + Iterate for less than the width

Read the tile ID (four bytes)

Create a tileset ID variable, intialised to zero.

If the ID is not equal to –1 (empty tile)

Iterate for less than the tileset count

If the tile ID is greater than the current tileset’s tile count (calculated from the image width and height and the tile size), subtract the tile count from the id, and add one to the tileset ID. This will transform the tile ID from a “global” ID to one that is specific to the current tileset. Otherwise, break out of the loop.

Set the tile buffer at index x + y \* width to a new tile with the calculated tile ID and the tileset ID.

An additional tile layer named “agent mask” will define where the agents can and cannot walk. It will be loaded into an array of Booleans, when the tile ID is equal –1 (empty tile) it will be false, and otherwise the array will contain and entry that is true.

Another object layer will be created called “items”. It will have the key’s location.

There will also be an object layer called “doors”. It will have a rectangle containing the trigger where the player can touch to open the door and win the game.

Only a single level is planned.

11.0 Interface

There will be a basic HUD showing the player whether or not they have a key, and whether or not they have been spotted.

11.1 Menu

The menu will have two options, “play” and “quit”.

11.2 Camera

The camera will be a 2D top-down view that follows the player.

11.3 Controls

WASD keys for movement.

14.0 Asset List

The sprites included in the 16x16 Dungeon Tileset.

16.0 Technical Risks

The AI might be too slow using a grid-based system, so a one using nodes or a navmesh might have to be implemented. As long as the amount of agents in a given level is kept to a minimum, this won’t be a problem. An optimisation that could be implemented is “sleeping” the agents that are off-screen so they don’t use up processing power.

Schedule:

* Get basic sprites drawing on the screen (day 1)
* Get levels loading from a binary file (day 1)
* Get AI agents pathfinding a point on the level (day 2 & 3)
* Add player movement and the door unlock mechanic (day 3)
* Make AI agents able to spot the player and transition states accordingly (day 4 & 5)
* Add a menu and UI (day 5)