A browser-based roguelike game: Project Skirmish

1. Introduction

Roguelike games, taking inspiration from the 1980 title "Rogue," are recognised for their procedural level generation and turn-based mechanics. One of the advantages that procedural generation offers over manual level design is the creation of a dynamic gameplay environment, which provides greater replayability for the player (and developer) by providing a unique experience each time.

This ongoing project known as Skirmish, started as a way to broaden my skills and understanding of the TypeScript programming language. Initially building from a tutorial using the Roguelike Toolkit JavaScript (ROT.js) then extending its functionality by incorporating tile graphics and animations through the integration of PIXI.js, a JavaScript library that supports both 2D and 3D graphics and rendering.

The choice of TypeScript as the foundational language was driven by the popularity of JavaScript and having a desire to learn how to create a game that would seamlessly run in web browsers. This project has been a journey of personal growth and continuous learning, demonstrating my commitment to self-improvement and exploration within the world of game development – and programming in general.

With this learning journey, I have two fundamental objectives. Firstly, I aim to showcase my proficiency in TypeScript and game development, highlighting the dedication I bring to personal projects. Secondly, I aspire to lay the groundwork for potential future extensions, with consideration to open-source the code in the future in which might help or inspire other developers

2. Project Description

JavaScript stands as one of the industry's favoured languages, primarily due to its versatility in web development. Its reach across various browsers and dynamic capabilities has made it one of the most popular languages of 2023. Building upon this foundation is TypeScript, an extension of JavaScript that adds static typing. With TypeScript, developers gain an enhanced tool that allows for more streamlined and error-resistant coding, making it a natural choice for larger scale projects.

One of the pivotal tools aiding this project's progression is the ROT.js toolkit. Designed explicitly for roguelike games, ROT.js acts as a catalyst, expediting the development phase with its pre-coded features. This means developers can leap ahead to refining gameplay, instead of getting bogged down with foundational coding.

When diving into the world of roguelike development, one quickly realizes that there's a treasure trove of resources online. A plethora of documentation awaits the eager developer, and platforms like GitHub offer a deep dive into practical code examples. Notably, the 7DRL challenge, where developers craft a roguelike game in just seven days, stands as a testament to the genre's vibrant community and the shared drive to innovate.

But what's a game without its aesthetics? Traditional roguelikes, while nostalgic with their ASCII graphics, can often feel limited in visual appeal. By introducing tile-based graphics and animations, the gameplay transforms. The visuals become richer, and the player's experience is elevated, merging the classic with the contemporary.

Lastly, in this pursuit of graphical enhancement, the PIXI.js library emerges as the frontrunner for TypeScript. It's not just about rendering graphics; it's about doing so efficiently and beautifully. PIXI.js offers the perfect blend of performance and versatility, making it the top choice for this project.

3. Technologies Used

This project came together with careful deliberation on which tools and technologies to use. Some of these technologies include:

1. Typescript

TypeScript is a superset of JavaScript created by Microsoft in 2012. Developed by Anders Hejlsberg, its purpose is to address challenges in large-scale JavaScript applications by adding optional static types. This helps catch errors early in development. While enhancing JavaScript, TypeScript compiles down to it, ensuring wide compatibility and introducing features from recent ECMAScript standards.

2. Vite

Vite is a next-generation frontend tooling solution introduced by Evan You, the creator of Vue.js, in early 2020. Designed to improve the developer experience, Vite provides rapid cold-server starts and blazing-fast hot module replacement (HMR). Unlike traditional bundlers, Vite serves code over native ES modules, making it faster and more efficient. As it has grown in popularity, Vite has expanded its support beyond Vue, accommodating frameworks like React, Preact, and Svelte, and has become a go-to choice for developers looking for an optimized and modern build setup.

3. ROT.js

ROT.js is a library specifically designed for the creation of roguelike games in the browser. Developed by Ondřej Žár, ROT.js seeks to provide modern developers with tools to create games in this genre. ROT.js offers a wide array of features critical for roguelike development, such as map generation, pathfinding, and more. It simplifies the process of creating these games by providing pre-built modules that would otherwise take a considerable amount of time to code from scratch.

4. ROT.js Tutorial

The ROT.js tutorial by Nick Klepinger guides learners through creating a roguelike game using ROT.js, TypeScript, and ViteJS. It covers topics such as setting up a playable character, map generation, dungeon creation, enemy interaction, UI development, inventory management, among other things

5. PIXI.js

PIXI.js is a rendering engine that allows for the creation of visually rich interactive graphics and animations in the browser. It was created by Mat Groves and introduced around 2013. PIXI.js is renowned for its fast performance and flexibility. It's hardware-accelerated with WebGL, making it suitable for creating complex graphics, including games. However, it's also capable of falling back to HTML5's canvas if needed, ensuring broad compatibility. Developers often choose PIXI.js for its ability to handle 2D graphics efficiently and its user-friendly API.

6. Codeanywhere

Founded in 2013, Codeanywhere started as a cloud-based code editor that aimed to provide developers a platform to write, edit, and collaborate on code from any device with an internet connection. Codeanywhere offers an integrated development environment (IDE) in the cloud. Developers can access their projects and code from anywhere, whether on a desktop, tablet, or even a mobile phone. Features include collaboration tools, where multiple users can code in real-time, similar to how one might collaborate on a Google Doc. The platform supports multiple programming languages and offers features like code completion, error checking, and more. It's particularly popular among developers who require a versatile coding environment, especially when collaborating with teams distributed across different locations.

7. Itch.io and IknowKingRabbit

The graphics used in this project were created by Aleksandr Makarov aka IKnowKingRabbit, who provides the assets to be used, while these graphics are not free (something that would need to be changed if I decide to release the code as open source), they are licensed to be used freely for both commercial and non-commercial purposes. The graphics themselves were purchased off itch.io.

itch.io was established in 2013 by Leaf Corcoran, providing an accessible platform for indie game developers to distribute, sell, and showcase their games and creative content. Beyond just games, it has become a valuable resource for game art assets, offering a diverse range of graphics, sprites, textures, and other design elements that developers can use to enhance their projects, with flexible pricing options including pay-what-you-want, which encourages the sharing and accessibility of creative resources within the indie game development community.

4. Beyond the Tutorial: Custom Enhancements

Rendering with PIXI.JS

The approach taken to include PIXI.js in the project involved following along the documentation from the PIXI.JS website to learn how to do it. With PIXI.js everything gets rendered in a PIXI.Application and this is instantiated in the main.ts class like so:

```
16
17  const app = new PIXI.Application({
18      width: 1140,
19      height: 768,
20      backgroundColor: 0x242234,
21      antialias: false,
22      resolution: 1,
23      autoDensity: true
24  });
25
```

(Fig.1, creating a new PIXI.Application)

With the PIXI.Application defined as app (Fig.1) it can then be passed through as a parameter to the the rest of the classes within the program.

In the game-map.ts class, exists the render method, which is used to draw the ASCII characters to the screen using the functions from the ROT.js library. The proceduarally generated tiles, stored in an array for the map is then drawn to the screen which has been well explained within Nick Klepinger's guide. It is within this class that the rendering happens Using the PIXI.js library.

A new class was created; sprite-manager.ts which defines the tilesetData along with textures that are parsed and loaded into memory. This sprite-manager class also contains a defined stage or PIXI.Container named as camera, which is used to render all of the tile and sprite animations onto the display. A camera container is used because unlike the movement in the original tutorial, in my implementation I have decided to go with a centered camera viewport.

With an instance of the sprite-manager it can be passed throughout the application as needed whenever graphic rendering needs to happen. In this case on the game-map.ts class within the render method, at the location where text from the ROT.js would normally draw ASCII characters on screen, PIXI.js functions are used to instead render sprites and animations to the screen. Because the tilesize of the sprites used for this project are 16x16 pixels a conversion from the font-sized point based system of the game-map needed to be done to correctly render the tiles on screen. One example of how this conversion works is in the code showcased below for centering the camera container to the player on screen.

Extending the procedural generation

The second major enhancement was adding extra tile types to the procedural generation for the game-map. This included:

- the addition of corner tiles for the rectangular rooms
- A door tile that would randomly be assigned to passageways that connect rooms
- Furnishing tiles, such as bookcases, tables and braziers

5. Implementation

Centering the view on the player

```
const TILESIZE = 16
const SPRITE_SCALE = 3
const MAP_SCALE = 2
const UI_MESSAGELOG = 128

if (this.spriteManager.camera) {

  let xOffset = window.engine.player.x * TILESIZE * SPRITE_SCALE
                - (Math.floor(Engine.MAP_WIDTH / 2) * TILESIZE * MAP_SCALE) + TILESIZE * MAP_SCALE;
  let yOffset = window.engine.player.y * TILESIZE * SPRITE_SCALE
                - (Math.floor(Engine.MAP_HEIGHT / 2) * TILESIZE * MAP_SCALE) - UI_MESSAGELOG;

                this.spriteManager.camera.position.set(-xOffset, -yOffset);

  const mask = new PIXI.Graphics();
  mask.beginFill(0xFFFFFF);
  mask.drawRect(62, 64, 710, 624);
  mask.endFill();
  this.spriteManager.camera.mask = mask;

}
```

(Fig.2, repositioning the camera to center on the player)

The approach decided for this was to render all the textures and sprites on their own PIXI.Container (camera) that the x and y location of the camera would be set by subtracting the calculated offset based on the players location.

Demonstrated above (Fig.2), firstly we can see the constants have been declared, such as the tilesize and scale. In addition to the rendering of the map, a mask has also been applied in this case to limit the viewable area of the rendered graphics. This is to account for areas on the screen where UI are present and which we do not want to display any rendered graphics.

Adding additional tile types to the map generation

```
export const BOOKCASE_TILE: Tile = {
  walkable: true,
  transparent: true,
  visible: false,
  seen: false,
  dark: { char: 'H', fg: '#646464', bg: '#000' },
  light: { char: 'H', fg: '#c8c8c8', bg: '#000' },
  type: 'furnishing',
  texture: '',
};

export const TABLE_TILE: Tile = {
  walkable: true,
  transparent: true,
  visible: false,
  seen: false,
  dark: { char: '_', fg: '#646464', bg: '#000' },
  light: { char: '_', fg: '#c8c8c8', bg: '#000' },
  type: 'furnishing',
  texture: '',
};
```

(Fig.3, creating new tile-types)

The model that configured the tile-types was extended to include the new tiles as shown above (Fig.3). Once the models were defined they can be brought through to the procgen.ts class. In here the methods could be expanded to include the new tiles.

```
buildRoom() {
  for (let y = 0; y < this.height; y++) {
    const row = new Array(this.width);
    for (let x = 0; x < this.width; x++) {
      if (x === 0 && y === 0) {
        row[x] = { ...WALL_TILE_TOP_LEFT };
      } else if (x === this.width - 1 && y === 0) {
        row[x] = { ...WALL_TILE_TOP_RIGHT };
      } else if (x === 0 && y === this.height - 1) {
        row[x] = { ...WALL_TILE_BOTTOM_LEFT };
      } else if (x === this.width - 1 && y === this.height - 1) {
        row[x] = { ...WALL_TILE_BOTTOM_RIGHT };
      } else if (y === 0) {
        row[x] = { ...WALL_TILE_TOP };
      } else if (y === this.height - 1) {
        row[x] = { ...WALL_TILE_BOTTOM };
      } else if (x === 0) {
        row[x] = { ...WALL_TILE_LEFT };
      } else if (x === this.width - 1) {
        row[x] = { ...WALL_TILE_RIGHT };
      } else {
        row[x] = { ...FLOOR_TILE, texture: (Math.floor(Math.random() * 5) + 1).toString()};
      }
    }
    this.tiles[y] = row;
  }
  if (!this.firstRoom) {
    // Place bookcases in larger rooms only
    if (this.width >= 6 && this.height >= 6) {
      this.placeBookcases();
    }
    this.placeTables();
    this.placeBraziers();
  }
}
```

(Fig.4, The buildRoom method)

For example (Fig.4), the buildRoom method was expanded to draw tiles in each of the four corners, in addition to this, as long as it is not the firstRoom (the room that the player starts in) then it will also call the placeBookcases(), placeTables() and the placeBraziers() methods.

placeTables

```
placeTables() {
  // Determine the size of the "middle region"
  const middleWidth = Math.floor(this.width * 0.5);
  const middleHeight = Math.floor(this.height * 0.5);

  // Determine the starting point of the "middle region"
  const startX = Math.floor((this.width - middleWidth) / 2);
  const startY = Math.floor((this.height - middleHeight) / 2);

  const numberOfTables = generateRandomNumber(1, 6); // Adjust as needed for the number of tables

  for (let i = 0; i < numberOfTables; i++) {
      const x = generateRandomNumber(startX, startX + middleWidth - 1);
      const y = generateRandomNumber(startY, startY + middleHeight - 1);

      // Check if the tile isn't already occupied by another entity, bookcase, or another table
      if (this.tiles[y][x].type === 'floor') {
          this.tiles[y][x] = { ...TABLE_TILE };
      } else {
          // decrement the counter to retry if the spot was occupied
          i--;
      }
  }
}
```

(Fig.5, the placeTables method)

placeTables (Fig.5) populates a grid's central 50% area with 1-6 tables, ensuring no overlaps with existing entities.

Detailed Breakdown

1. **Middle Region Calculation:** Determines the size and starting point of the middle region.
2. **Random Table Count:** Generates a random number of tables to place.
3. **Table Placement Loop:** Iterates through each table, generating random coordinates within the middle region.
4. **Occupancy Check and Placement:** Checks if the tile is a 'floor' tile before placement, retrying if occupied.

Summary

This method offers a concise and effective approach to randomly placing tables, promoting a balanced distribution while avoiding entity overlap.

placeBookcases

```
placeBookcases() {
  const numberOfBookcases = generateRandomNumber(1, 10); // Adjust as needed

  const isAdjacentToWall = (x: number, y: number) => {
      // Check tiles around the current position
      const directions = [
          { dx: 0, dy: -1 }, // Above
          { dx: 1, dy: 0 },  // Right
          { dx: 0, dy: 1 },  // Below
          { dx: -1, dy: 0 }  // Left
      ];

      for (const { dx, dy } of directions) {
          const adjX = x + dx;
          const adjY = y + dy;

          if (adjX >= 0 && adjX < this.width && adjY >= 0 && adjY < this.height) {
              if (this.tiles[adjY][adjX].type.startsWith('wall')) {
                  return true;
              }
          }
      }

      return false;
  };

  for (let i = 0; i < numberOfBookcases; i++) {
      const x = generateRandomNumber(1, this.width - 2); // Keep away from walls
      const y = generateRandomNumber(1, this.height - 2);

      // Check if the tile isn't already occupied by another entity or bookcase
      if (this.tiles[y][x].type === 'floor' && isAdjacentToWall(x, y)) {
          this.tiles[y][x] = { ...BOOKCASE_TILE };
      } else {
          // decrement the counter to retry if the spot was occupied or not adjacent to a wall
          i--;
      }
  }
}
```

(Fig.6, placeBookcases method)

placeBookcases, (Fig.6) places 1-10 bookcases on 'floor' tiles adjacent to walls within a grid, avoiding overlap with existing entities.

Detailed Breakdown

1. **Random Bookcase Count:** Determines how many bookcases to place, ranging from 1 to 10.

2. **Adjacency Check:** isAdjacentToWall(x, y): Checks if a given tile is next to a wall.
3. **Bookcase Placement Loop:** Iterates through the number of bookcases to be placed, generating coordinates within valid grid bounds.
4. **Placement Validation and Execution:** Checks if the tile is a 'floor' tile and adjacent to a wall before placing a bookcase, retrying if conditions are not met.

Summary

The method ensures bookcases are only placed next to walls, maintaining game aesthetics and design rules,

placeBraziers

```
placeBraziers() {
  const corners: [number, number][] = [
    [(this.x + 1), (this.y + 1)],                          // Top-left corner
    [this.x - 1 + this.width - 1, this.y + 1],             // Top-right corner
    [this.x + 1, this.y + this.height - 2],                // Bottom-left corner
    [this.x - 1 + this.width - 1, this.y + this.height - 2], // Bottom-right corner
  ];

  for (const [cornerX, cornerY] of corners) {
    // For each corner, independently check for a 25% chance
    if (Math.random() <= 0.6 && this.isPositionValidForBrazier(cornerX, cornerY) &&
        !this.isBookcasePresent(cornerX, cornerY)) {
          this.tiles[cornerY - this.y][cornerX - this.x] = { ...BRAZIER_TILE };
        }
  }
}

isPositionValidForBrazier(x: number, y: number): boolean {
  // Check if the tile is inside the room and is a floor tile
  if (x >= this.x && x < this.x + this.width && y >= this.y && y < this.y + this.height &&
      this.tiles[y - this.y][x - this.x].type === 'floor') {
    return true;
  }
  return false;
}

isBookcasePresent(x: number, y: number): boolean {
  // Check if there's a bookcase at the specified position
  return (
    x >= this.x &&
    x < this.x + this.width &&
    y >= this.y &&
    y < this.y + this.height &&
    this.tiles[y - this.y][x - this.x].type === 'bookcase'
  );
}
```

(Fig.7, placeBraziers method)

placeBraziers (Fig.7) attempts to place braziers at each corner of a room, adhering to specific conditions for placement.

Detailed Breakdown

1. **Corner Calculation:** Determines the four corners of the room.
2. **Brazier Placement Loop:** Iterates through each corner, attempting to place a brazier.
3. **Placement Conditions:** Checks for a 60% chance, valid position, and absence of a bookcase before placement.
4. **Position Validation:** isPositionValidForBrazier(x, y): Ensures the position is within the room and on a 'floor' tile.
5. **Bookcase Check:** isBookcasePresent(x, y): Verifies there’s no bookcase at the specified position.

Summary

This method enhances room aesthetics with potential brazier placement at corners, ensuring no conflicts with bookcases or invalid positions.

6. Conclusion

At the completion of this first phase of the Skirmish project, a proof-of-concept alpha version of the roguelike game has been produced. The integration of TypeScript, ROT.js, and PIXI.js has proven to be a synergistic choice, bringing together the robustness of static typing, the efficiency of roguelike-specific functionalities, and the visual flair of advanced rendering.

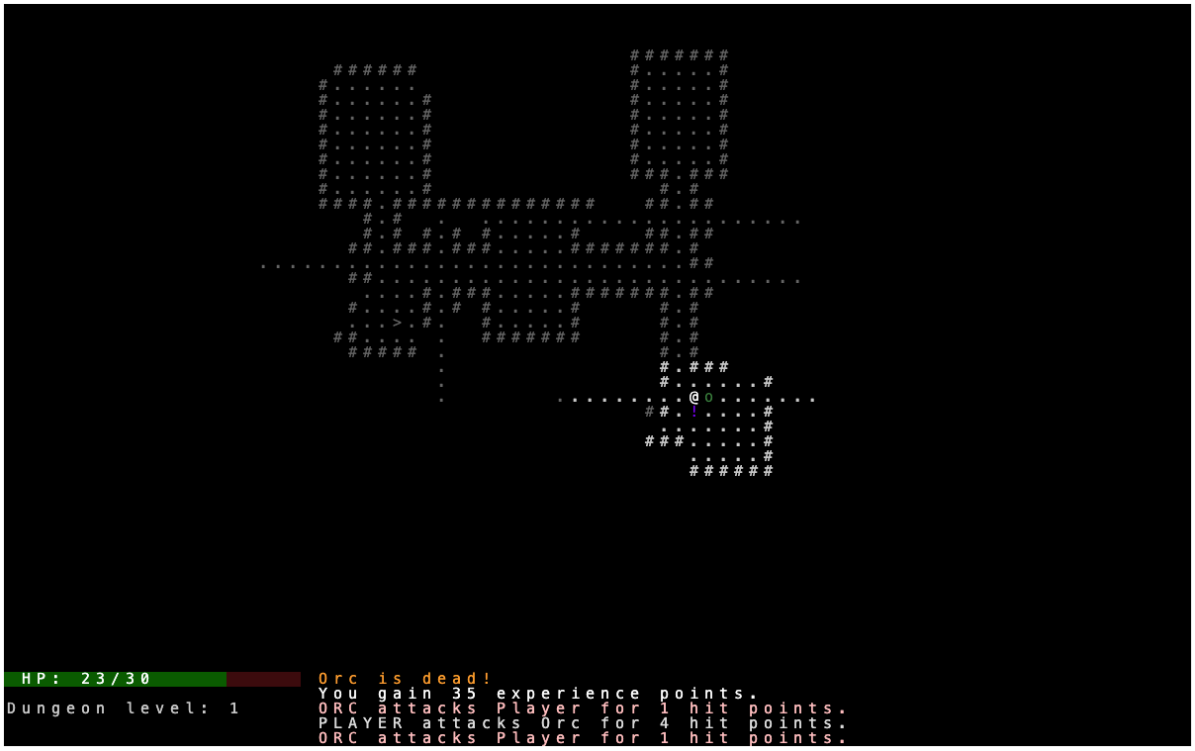
This journey has not only expanded my proficiency in TypeScript and game development but has also opened doors for future enhancements and iterations. The procedural generation enhancements and the graphical upgrades are testament to the project's potential for growth and depth. The custom enhancements made beyond the tutorial have imbued the game with a unique identity, setting the stage for further exploration and innovation.

As an ongoing project, Skirmish stands as a living, evolving entity. The current state, while functional and visually engaging, is merely the first step in a longer journey of development and learning. Future iterations will undoubtedly see more features, optimizations, and refinements, as the project continues to mature and evolve.

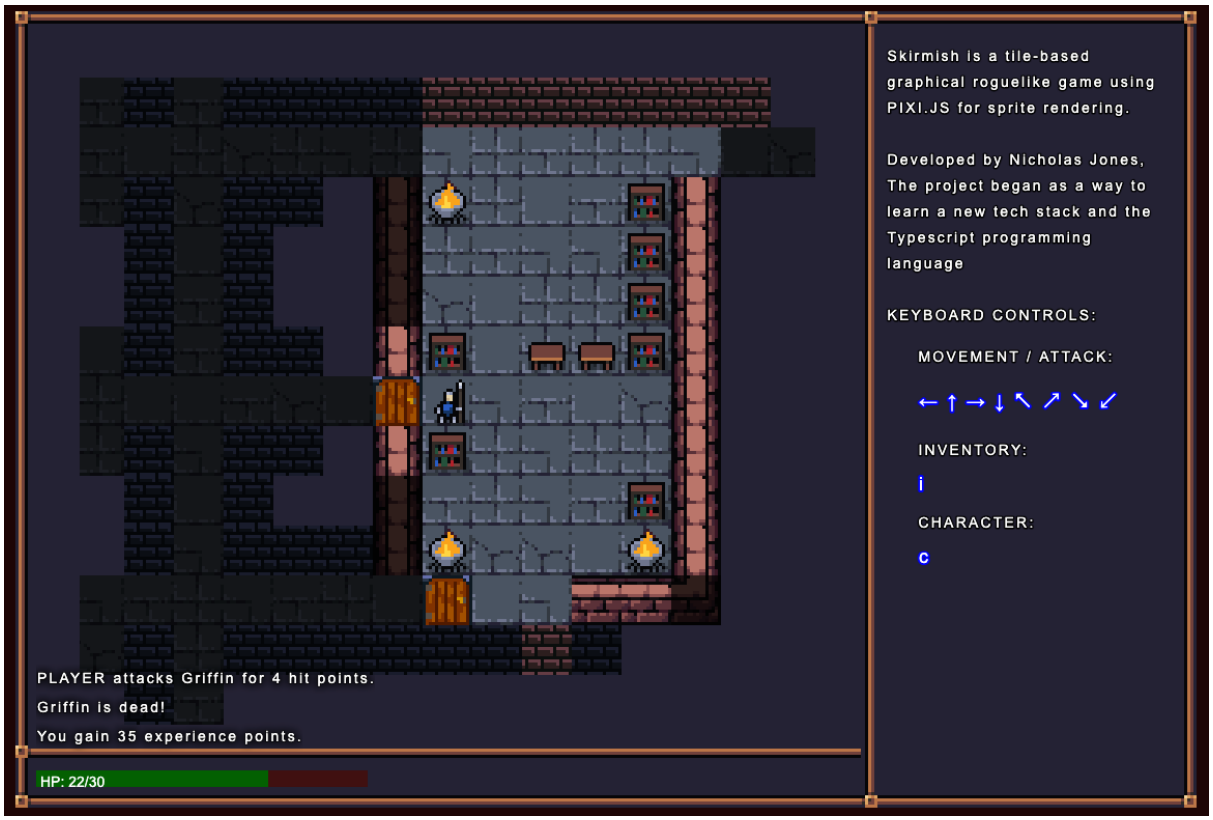
This documentation serves not just as a record of progress and a guide for future development, but also as a showcase of commitment, technical skill, and a passion for game development. It reflects a dedication to learning, a willingness to tackle challenges, and a drive to create engaging, dynamic gameplay experiences.

Skirmish, in its current form, is a solid stepping stone towards mastering the intricacies of game development, and a promising glimpse into the potential future of this roguelike adventure.

Appendix



A typical ASCII character level, from the ROT.js tutorial, before the Skirmish Project.



After Skrimish Proejct using sprite graphics and textures.