Code for Solo Play

By Peter Jones May 31, 2023 In Code, Games

Solo play, in the context of role-playing games (RPGs), refers to engaging in the game as a single player, without the presence of a game master or a group of other players. It allows individuals to enjoy RPG experiences on their own, taking on the roles of both the player character(s) and the game master.

Solo play provides a unique and immersive gaming experience where the player can create their own stories, make decisions, and explore game worlds at their own pace. It offers the flexibility to play whenever

desired, without the need to coordinate schedules or find a group of players.

To facilitate solo play, various resources and tools have been developed. These include rule systems designed specifically for solo adventures, game master emulators that simulate the decision-making of a game master, random generators for generating encounters and events, and solo-focused adventures or modules.

Solo play can be a rewarding experience for players who enjoy self-directed storytelling, tactical challenges, character development, and exploration of rich game worlds. It allows for personal creativity, deep immersion, and the ability to adapt the game experience to individual preferences and play styles.

> **“Remember, the most important aspect of solo play is to have fun and enjoy the experience. Feel free to experiment, adjust rules as needed, and create a gaming experience that suits your preferences.”**

Adapting existing guides for solo play.

Here are some tips and ideas for adapting existing RPG rules for solo play:

- **Choose a solo-friendly RPG system:** Some RPG systems are specifically designed for solo play or offer rule sets that are easily adaptable. Look for systems like Ironsworn, Mythic Game Master Emulator, or the Solo Adventurer’s Toolbox. These systems often include mechanisms to generate random events, NPCs, and quests.

- **Create a character:** Develop a character concept and build their stats and abilities according to the rules of the RPG system you're using. Consider your character's strengths, weaknesses, and backstory to make the solo experience more engaging.
- **Modify encounters and challenges:** In a traditional RPG, encounters and challenges are typically designed for a group of players. When playing solo, you may need to adjust the difficulty level. Consider reducing the number or strength of opponents or adjusting the mechanics to compensate for the lack of a full party.
- **Use random generators:** Random generators can be a valuable tool for solo play. They can help you generate NPCs, quests, dungeons, and other elements of the game world. You can find online generators or create your own tables based on the setting and themes of your RPG.
- **Create a GM emulator:** If your chosen RPG system doesn't have a built-in Game Master emulator, you can create your own. Use a set of yes/no questions or dice rolls to determine the outcomes of your character's actions and to simulate the decisions a Game Master would make.
- **Keep a journal:** Document your character's progress, decisions, and the outcomes of their actions. This can help you keep track of the story, maintain continuity, and provide a sense of accomplishment as you see your character's growth and development over time.
- **Experiment with solo modules or adventures:** Some RPG systems offer solo modules or adventures designed specifically for one player. These can provide structured narratives, quests, and encounters tailored to solo play.
- **Embrace improvisation:** Solo play gives you the freedom to explore and make decisions without the constraints of a group. Embrace the opportunity to improvise and shape the story according to your character's choices.

Solo Play Guides

If that sound like hard work, then you have the option of using a predefined rule system. Here are some published solo play guides, rules, and modules for role-playing games along with their descriptions, authors, publishers and publication dates:

- **Mythic Game Master Emulator** by Tom Pigeon (Publisher: Word Mill Games, 2006): Mythic is a system-agnostic toolkit that allows you to play any role-playing game in solo mode. It provides a set of rules and tables to generate random events, determine outcomes, and simulate the role of the Game Master. It offers flexibility and support for creating your own solo adventures.
- **Scarlet Heroes** by Kevin Crawford (Publisher: Sine Nomine Publishing, 2014): Scarlet Heroes is a complete role-playing game designed specifically for solo play or for groups with a single player and Game Master. It focuses on classic fantasy adventures and offers rules and tools tailored for a solo experience. The game includes guidelines for adapting existing modules for solo play.
- **Mythic Variations** by Tana Pigeon (Publisher: Word Mill Games, 2014): Mythic Variations is an expansion to the Mythic Game Master Emulator system. It introduces new variations and options for solo play, including additional charts and rules for generating more complex events, character arcs, and story developments. It expands the possibilities for solo role-playing.
- **Four Against Darkness** by Andrea Sfiligoi (Publisher: Ganesha Games, 2017): Four Against Darkness is a solitaire dungeon-delving game that uses a simple set of rules and tables. It allows you to create a party of adventurers and explore dungeons, fight monsters, and discover treasure. The game includes a variety of scenarios and provides a quick and accessible solo gaming experience.
- **Solo Adventurer's Toolbox** by Paul Bimler (Publisher: Zozer Games, 2017): The Solo Adventurer's Toolbox is a supplement for the Cepheus Engine role-playing game, but it can be adapted to other systems as well. It provides resources and techniques for playing solo, including tools for generating encounters, events, and NPC

reactions. The toolbox helps create a dynamic and engaging solo experience.

- **Ironsworn** by Shawn Tomkin (Publisher: Shawn Tomkin, 2018): Ironsworn is a role-playing game that is designed for solo play or cooperative play with a group. It features a dark fantasy setting and provides rules and tools to guide players through quests and adventures. The game mechanics use a combination of moves and narrative prompts to drive the story forward.

These are just a few examples of published solo play guides, rules, and modules available. Each of these resources offers different approaches to solo play, so you can choose the one that aligns best with your preferences and the RPG system you want to play.

System Reference Documents (SRDs)

The System Reference Document (SRD) for role-playing games typically refers to the open gaming content and rules released under the Open Game License (OGL). The SRD provides a subset of rules and content that can be freely used and referenced by game designers and developers. This can be useful starting point to adopting solo play.

The specific SRD content may vary depending on the game system or edition. Here are references to some popular SRDs:

1. Dungeons & Dragons 5th Edition SRD:
   - Website: https://www.5esrd.com/
   - GitHub Repository: https://github.com/thebombzen/srd/
2. Pathfinder RPG SRD:
   - Website: https://www.d20pfsrd.com/
   - Archives of Nethys: https://2e.aonprd.com/
3. OpenD6 SRD:
   - Website: http://www.opengamingfoundation.org/d6/

4. Stars Without Number SRD:
    - Website: https://www.drivethrurpg.com/product/230009/Stars-Without-Number-Revised-Edition-Free-Version

Please note that the availability and content of SRDs may change over time. It's always recommended to verify the current sources and licenses for the specific game system you are interested in.

Code for Random Generators

Using code to assist with solo play RPGs can provide several benefits:

- **Automation:** Code can automate various aspects of the game, such as randomizing encounters, generating NPCs, resolving combat, or managing game mechanics. This automation saves time and effort by handling repetitive tasks, allowing you to focus more on the storytelling and decision-making aspects of the game.
- **Rule Adherence**: By using code, you can ensure consistent and accurate application of game rules. The code can enforce rules, calculate probabilities, and handle complex mechanics, reducing the likelihood of errors or oversights in gameplay.
- **Randomization:** Code can generate random elements, such as random encounters, loot, or events, adding unpredictability and variety to your solo game sessions. This randomness can enhance the immersion and challenge of the game.
- **Solo Game Structures:** Code can help create structures and frameworks specific to solo play, such as generating storylines, managing character progression, or providing prompts for decision-making. These structures provide a framework for solo play and can enhance the overall experience.
- **Flexibility and Customization:** Code allows you to customize and adapt the game mechanics to fit your specific preferences and playstyle. You can modify existing code or create your own scripts to tailor the game experience to your liking.

- **Visualization:** Code can be used to create visual representations of game elements, such as maps, character sheets, or interactive interfaces. These visualizations can enhance the immersion and make it easier to understand and navigate the game world.

Overall, using code to assist with solo play RPGs provides automation, rule adherence, randomization, customized game structures, flexibility, and visualization. It can enhance your solo gaming experience by streamlining processes, providing dynamic content, and enabling a more immersive and interactive gameplay environment.

Getting Started

- Intro to Python
- Details on p5.js

Dice Roll

Here's an example of code that allows you to roll various types of dice (d4, d6, d8, etc.) with input in the format of "NdX + Y":

```
# python - Dice Roll with Modifiers

import random

def roll_dice(dice_string):
    # Split the input string into the number of dice, dice type, and
modifier
    parts = dice_string.split("d")
    num_dice = int(parts[0])

    # Check if a modifier is present
    if "+" in parts[1]:
        dice, modifier = parts[1].split("+")
        modifier = int(modifier.strip())
    elif "-" in parts[1]:
        dice, modifier = parts[1].split("-")
        modifier = -int(modifier.strip())
    else:
        dice = parts[1]
```

```
            modifier = 0

        dice_type = int(dice)

        # Roll the dice
        rolls = [random.randint(1, dice_type) for _ in range(num_dice)]

        # Calculate the total result
        total = sum(rolls) + modifier

        # Print the individual rolls and the total result
        print(f"Rolls: {rolls}")
        print(f"Total: {total}")
```

You can use this function by calling roll_dice() with a dice string as the argument. Here are some examples:

```
roll_dice("4d6 + 2")  # Roll four six-sided dice and add 2 to the total
roll_dice("1d8 - 1")  # Roll one eight-sided die and subtract 1 from the
total
roll_dice("2d4")      # Roll two four-sided dice without any modifier
```

Please feel free to modify the code as per your specific requirements or incorporate it into a larger program.

Grid of Numbers

Here’s an example code that generates a uniform grid of numbers for dice rolls and formats it for printing on A4/US letter size:

```
#python - Grid of Numbers

def generate_dice_grid(dice_expression, rows, columns):
    # Calculate the maximum value based on the dice expression
    dice_max = int(dice_expression.split("d")[-1]) +
int(dice_expression.split("d")[0]) - 1
```

```
        # Create the grid of numbers
        grid = []
        for i in range(rows):
            row = []
            for j in range(columns):
                value = i * columns + j + 1
                if value <= dice_max:
                    row.append(value)
                else:
                    row.append(None)
            grid.append(row)

        return grid

    def print_dice_grid(grid):
        max_value_length = len(str(grid[-1][-1])) + 2
        for row in grid:
            for value in row:
                if value is None:
                    print(" " * max_value_length, end=" ")
                else:
                    print(f"{value:>{max_value_length}}", end=" ")
            print()

    # Example usage
    dice_expression = "4d6 + 2"
    rows = 6
    columns = 8

    grid = generate_dice_grid(dice_expression, rows, columns)
    print_dice_grid(grid)
```

In this code, the generate_dice_grid function takes the dice expression (e.g., “4d6 + 2”), the number of rows, and the number of columns as input. It calculates the maximum value based on the dice expression and generates a grid of numbers. The numbers in the grid are populated based on their position and the maximum value.

The print_dice_grid function formats and prints the grid, ensuring that the numbers are aligned properly. It calculates the maximum value length in the grid and pads the numbers accordingly.

You can modify the dice_expression, rows, and columns variables in the example usage to customize the grid based on your requirements.

Adventure Outline

Here's an example of code for generating an adventure outline. This code provides a basic structure for an adventure, including a quest, NPCs, locations, and encounters:

```
#python - Code to generate adventure outline

import random

class AdventureGenerator:
    quests = ["Retrieve an artifact", "Rescue a captive", "Slay a
monster", "Uncover a secret", "Deliver an important message"]
    locations = ["Ancient ruins", "Enchanted forest", "Mysterious
caverns", "Haunted castle", "Lost city"]
    NPCs = ["Mysterious wizard", "Skilled rogue", Wise old sage", "Brave
knight", "Shady merchant"]

    @staticmethod
    def generate_adventure():
        adventure = {}
        adventure["quest"] = random.choice(AdventureGenerator.quests)
        adventure["location"] =
random.choice(AdventureGenerator.locations)
        adventure["npc"] = random.choice(AdventureGenerator.NPCs)
        adventure["encounters"] =
AdventureGenerator.generate_encounters()
        return adventure

    @staticmethod
    def generate_encounters():
        num_encounters = random.randint(3, 6)
        encounters = []
        for _ in range(num_encounters):
            encounter = {
                "location": random.choice(AdventureGenerator.locations),
                "npc": random.choice(AdventureGenerator.NPCs),
                "description": "A challenge awaits..."
            }
            encounters.append(encounter)
        return encounters
```

```
    # Example usage:

    adventure = AdventureGenerator.generate_adventure()

    print("Adventure Outline:")
    print("Quest:", adventure["quest"])
    print("Location:", adventure["location"])
    print("NPC:", adventure["npc"])
    print("Encounters:")
    for i, encounter in enumerate(adventure["encounters"]):
        print(f"\nEncounter {i+1}:")
        print("Location:", encounter["location"])
        print("NPC:", encounter["npc"])
        print("Description:", encounter["description"])
```

In the code above, the AdventureGenerator class provides a static method generate_adventure() that generates an adventure outline. It randomly selects a quest, location, and NPC from predefined lists. It also calls the generate_encounters() method to create a list of encounters associated with the adventure.

The generate_encounters() method determines a random number of encounters (between 3 and 6) and creates encounter objects with randomly chosen locations, NPCs, and a generic description.

The example usage demonstrates how to generate an adventure outline using the generate_adventure() method and prints the generated adventure's details, including the quest, location, NPC, and a list of encounters.

You can expand upon this code and add more details, customizations, or additional components to the adventure outline generator based on your specific requirements and the complexity of your selected RPG system.

Generate Character

Here's an example code to generate a basic OSR (Old School Renaissance) character using the System Reference Document (SRD) as a reference:

```python
# python - Generate Character

import random

# Character classes and their hit dice
classes = {
    "Fighter": "d8",
    "Cleric": "d6",
    "Thief": "d4",
    "Magic-User": "d4"
}

# Ability scores and their modifiers
abilities = {
    "Strength": 0,
    "Dexterity": 0,
    "Constitution": 0,
    "Intelligence": 0,
    "Wisdom": 0,
    "Charisma": 0
}

def roll_dice(dice):
    rolls, sides = map(int, dice.split("d"))
    return sum(random.randint(1, sides) for _ in range(rolls))

def generate_character():
    # Roll ability scores
    for ability in abilities:
        abilities[ability] = roll_dice("3d6")

    # Randomly select a character class
    character_class = random.choice(list(classes.keys()))

    # Generate hit points based on character class hit dice
    hit_dice = classes[character_class]
    hit_points = roll_dice(hit_dice)

    # Print the generated character
    print("Character Class:", character_class)
    print("Ability Scores:")
    for ability, score in abilities.items():
        print(ability + ":", score)
    print("Hit Points:", hit_points)
```

```
# Generate a character
generate_character()
```

In this code, we have a dictionary classes that defines the available character classes and their associated hit dice. The abilities dictionary represents the ability scores of the character.

The roll_dice function simulates rolling dice based on the provided dice notation (e.g., “3d6” for rolling three six-sided dice).

The generate_character function randomly selects a character class, rolls ability scores, and generates hit points based on the selected class’s hit dice. It then prints out the generated character’s class, ability scores, and hit points.

You can customize and expand upon this code by adding more options for character classes, incorporating additional character attributes, or including other elements from the SRD as per your requirements.

NPC Generator

Here’s an example of code for generating NPCs (Non-Player Characters) with race, class, stats, armor, weapon, and likely response:

```
# python - NPC Generator

import random

class NPCGenerator:
    races = ["Human", "Elf", "Dwarf", "Orc", "Goblin"]
    classes = ["Warrior", "Mage", "Rogue", "Cleric"]
    armor_types = ["Leather", "Chainmail", "Plate"]
    weapon_types = ["Sword", "Axe", "Bow", "Staff", "Dagger"]
    likely_responses = ["Friendly", "Neutral", "Hostile"]

    @staticmethod
    def generate_npc():
        npc = {}
```

```
            npc["race"] = random.choice(NPCGenerator.races)
            npc["class"] = random.choice(NPCGenerator.classes)
            npc["stats"] = {
                "Strength": random.randint(1, 10),
                "Dexterity": random.randint(1, 10),
                "Intelligence": random.randint(1, 10),
                "Wisdom": random.randint(1, 10),
                "Charisma": random.randint(1, 10)
            }
            npc["armor"] = random.choice(NPCGenerator.armor_types)
            npc["weapon"] = random.choice(NPCGenerator.weapon_types)
            npc["likely_response"] =
    random.choice(NPCGenerator.likely_responses)

            return npc

    # Example usage:

    npc = NPCGenerator.generate_npc()
    print("Race:", npc["race"])
    print("Class:", npc["class"])
    print("Stats:", npc["stats"])
    print("Armor:", npc["armor"])
    print("Weapon:", npc["weapon"])
    print("Likely Response:", npc["likely_response"])
```

In the code above, the NPCGenerator class provides a static method generate_npc() that generates a random NPC. It selects a race, class, and likely response from predefined lists. The stats are randomly generated within a range, and the armor and weapon types are chosen randomly as well.

You can modify the predefined lists (races, classes, armor_types, weapon_types, likely_responses) to include additional options or customize them according to your RPG system's rules and setting.

You can expand upon this code and add more features or details to the NPC generation based on your specific requirements.

Character Sheet

Here’s an example code that generates a character sheet in Markdown (MD) format:

```
#python - character sheet

def generate_character_sheet(character):
    sheet = f"# Character Sheet: {character['name']}\n\n"
    sheet += f"**Race:** {character['race']}\n\n"
    sheet += f"**Class:** {character['class']}\n\n"
    sheet += f"**Level:** {character['level']}\n\n"
    sheet += f"**Attributes:**\n\n"
    for attr, value in character['attributes'].items():
        sheet += f"- {attr.capitalize()}: {value}\n"
    sheet += "\n"
    sheet += f"**Skills:**\n\n"
    for skill, rank in character['skills'].items():
        sheet += f"- {skill.capitalize()}: {rank}\n"
    sheet += "\n"
    sheet += f"**Inventory:**\n\n"
    for item in character['inventory']:
        sheet += f"- {item}\n"
    return sheet

# Example character data
character_data = {
    "name": "Gandalf",
    "race": "Human",
    "class": "Wizard",
    "level": 10,
    "attributes": {
        "strength": 12,
        "dexterity": 10,
        "constitution": 14,
        "intelligence": 18,
        "wisdom": 16,
        "charisma": 14
    },
    "skills": {
        "arcana": 8,
        "history": 6,
        "persuasion": 4
    },
    "inventory": ["Staff", "Spellbook", "Potion of Healing"]
}

# Generate character sheet
character_sheet = generate_character_sheet(character_data)
```

```
# Print or save the character sheet
print(character_sheet)
```

In this code, the generate_character_sheet function takes a character dictionary as input and constructs a character sheet in Markdown format. It extracts the relevant information from the character data and formats it using Markdown syntax.

The example character data includes attributes, skills, and inventory information. You can modify the character data structure and add or remove fields as needed to match your RPG system or character sheet requirements.

The generated character sheet is stored in the character_sheet variable and can be printed or saved to a file.

Feel free to customize the code further based on your specific character sheet format and additional information you want to include.

GM Simulator

Here is code that provides a numbered list of options for the questions, incorporates weighting for yes and no responses based on difficulty parameters, and uses a d20 roll system where 1 is always a fail (no) and 20 is always a pass (yes):

```
# python - GM Simulator

import random

def ask_numbered_question(question, options):
    print(question)
    for i, option in enumerate(options):
        print(f"{i+1}. {option}")
    while True:
        response = input("Enter the number of your choice: ")
        if response.isdigit() and 1 <= int(response) <= len(options):
```

```
            return int(response)

def roll_d20():
    return random.randint(1, 20)

def simulate_game_master(difficulty):
    # Introduction
    print("Welcome to the Game Master Emulator!")
    print("You can simulate the decisions of a Game Master using this
tool.")

    # Main loop
    while True:
        # Prompt for player's action
        print("\nWhat do you want to do?")
        action = input("> ")

        # Simulate Game Master decision
        yes_weight = 10 + difficulty  # Adjust the weights based on
difficulty
        no_weight = 10 - difficulty

        if roll_d20() <= yes_weight:
            print("The action is successful.")
        else:
            print("The action failed.")

        if roll_d20() > no_weight:
            print("Something unexpected happens.")

        if roll_d20() > no_weight:
            print("Random encounter!")

        if roll_d20() <= yes_weight:
            print("You find valuable items or treasure.")

        if roll_d20() <= yes_weight:
            print("You receive useful information.")

        if roll_d20() > no_weight:
            print("There are obstacles in your path.")

        if roll_d20() <= yes_weight:
            skill_check_result = roll_d20()
            print("You rolled a", skill_check_result, "on the skill
check.")

        if roll_d20() > no_weight:
            print("You are in immediate danger.")

        # Prompt to continue or exit
```

```
            if not ask_numbered_question("Continue playing?", ["Yes", "No"])
    == 1:
                print("Exiting the Game Master Emulator.")
                break

    # Run the Game Master emulator
    difficulty = ask_numbered_question("Select difficulty:", ["Easy",
    "Medium", "Hard"])
    simulate_game_master(difficulty)
```

In this updated code, the ask_numbered_question function takes a question and a list of options. It displays the question along with the numbered options and returns the user's selected option as a number.

The roll_d20 function simulates rolling a d20, where the result is a random number between 1 and 20.

The simulate_game_master function now includes a difficulty parameter. The weights for yes and no responses are adjusted based on the difficulty level.

The emulator uses the ask_numbered_question function for the "Continue playing?" prompt, allowing the player to choose between "Yes" and "No" options.

Feel free to further customize the code according to your RPG scenario, including adding more options, adjusting the weighting system, or incorporating additional game mechanics.

Combat Resolution

Here's an example of code for a simple combat resolution between a solo character and an NPC, with inputs from the user per round:

```
    # python - Combat Resolution
```

```
import random

class Character:
    def __init__(self, name, health, attack_damage, defense):
        self.name = name
        self.health = health
        self.attack_damage = attack_damage
        self.defense = defense

    def attack(self):
        return random.randint(1, self.attack_damage)

    def take_damage(self, damage):
        self.health -= max(0, damage - self.defense)

def combat_resolution(player, npc):
    round_count = 1

    while player.health > 0 and npc.health > 0:
        print(f"\nRound {round_count} - {player.name} vs {npc.name}")
        print(f"{player.name} Health: {player.health} | {npc.name}
Health: {npc.health}")

        player_attack = player.attack()
        npc_attack = npc.attack()

        print(f"{player.name} attacks {npc.name} and deals
{player_attack} damage.")
        npc.take_damage(player_attack)

        if npc.health <= 0:
            print(f"{npc.name} has been defeated!")
            break

        print(f"{npc.name} attacks {player.name} and deals {npc_attack}
damage.")
        player.take_damage(npc_attack)

        if player.health <= 0:
            print(f"{player.name} has been defeated!")
            break

        round_count += 1

# Example usage:

player_name = input("Enter the name of your character: ")
player_health = int(input("Enter the health of your character: "))
player_attack_damage = int(input("Enter the attack damage of your
character: "))
player_defense = int(input("Enter the defense of your character: "))
```

```
    npc_name = input("Enter the name of the NPC: ")
    npc_health = int(input("Enter the health of the NPC: "))
    npc_attack_damage = int(input("Enter the attack damage of the NPC: "))
    npc_defense = int(input("Enter the defense of the NPC: "))

    player = Character(player_name, player_health, player_attack_damage,
    player_defense)
    npc = Character(npc_name, npc_health, npc_attack_damage, npc_defense)

    combat_resolution(player, npc)
```

In the code above, the Character class represents a character in the combat scenario. It has attributes such as name, health, attack damage, and defense. The attack() method randomly generates an attack value within the character's attack damage range, and the take_damage() method reduces the character's health based on the incoming damage, subtracting the defense value.

The combat_resolution() function takes a player character and an NPC as parameters. It loops through rounds until either the player or the NPC's health reaches zero. In each round, it displays the current health of both characters and their attacks. After each attack, it checks if either character's health has reached zero and breaks the loop if so.

The example usage prompts the user to enter the details of the player character and the NPC. The combat resolution is then initiated by calling the combat_resolution() function with the player and NPC instances.

Feel free to modify the code to suit your specific needs, add additional features, or enhance the combat mechanics based on your RPG system's rules.

Generating a Map

Here's an example of how you can generate a player map for an RPG with markers for a journey, random encounters, and destinations using p5.js:

```
let mapSize = 10;
let tileSize = 50;
let playerX = 0;
let playerY = 0;
let journeyPath = [];
let randomEncounters = [];
let destination;

function setup() {
  createCanvas(mapSize * tileSize, mapSize * tileSize);

  // Generate random journey path
  generateJourney();

  // Generate random encounters
  generateRandomEncounters();

  // Set a random destination
  destination = createVector(floor(random(mapSize)),
floor(random(mapSize)));
}

function draw() {
  background(220);

  // Draw map tiles
  for (let y = 0; y < mapSize; y++) {
    for (let x = 0; x < mapSize; x++) {
      let xPos = x * tileSize;
      let yPos = y * tileSize;

      // Draw journey path
      if (isInJourneyPath(x, y)) {
        fill(255, 255, 0);
        rect(xPos, yPos, tileSize, tileSize);
      }

      // Draw random encounters
      if (isRandomEncounter(x, y)) {
        fill(255, 0, 0);
        ellipse(xPos + tileSize / 2, yPos + tileSize / 2, tileSize / 2);
      }

      // Draw destination
      if (x === destination.x && y === destination.y) {
        fill(0, 255, 0);
        rect(xPos, yPos, tileSize, tileSize);
      }
```

```
      }
    }

    // Draw player
    let playerPosX = playerX * tileSize + tileSize / 2;
    let playerPosY = playerY * tileSize + tileSize / 2;
    fill(0, 0, 255);
    ellipse(playerPosX, playerPosY, tileSize / 2);
  }

  function keyPressed() {
    // Move player based on arrow keys
    if (keyCode === UP_ARROW && playerY > 0) {
      playerY--;
    } else if (keyCode === DOWN_ARROW && playerY < mapSize - 1) {
      playerY++;
    } else if (keyCode === LEFT_ARROW && playerX > 0) {
      playerX--;
    } else if (keyCode === RIGHT_ARROW && playerX < mapSize - 1) {
      playerX++;
    }

    // Check for encounters and destination
    checkEncounter();
    checkDestination();
  }

  function isInJourneyPath(x, y) {
    for (let i = 0; i < journeyPath.length; i++) {
      if (x === journeyPath[i].x && y === journeyPath[i].y) {
        return true;
      }
    }
    return false;
  }

  function isRandomEncounter(x, y) {
    for (let i = 0; i < randomEncounters.length; i++) {
      if (x === randomEncounters[i].x && y === randomEncounters[i].y) {
        return true;
      }
    }
    return false;
  }

  function checkEncounter() {
    if (isRandomEncounter(playerX, playerY)) {
      // Perform random encounter logic
      console.log("Random encounter!");
    }
  }
```

```
function checkDestination() {
  if (playerX === destination.x && playerY === destination.y) {
    // Perform destination reached logic
    console.log("Destination reached!");
  }
}

function generateJourney() {
  journeyPath = [];

  let numJourneyTiles = floor(random(mapSize / 2, mapSize - 1));
  let startX = 0;
  let startY = 0;

  for (let i = 0; i < numJourneyTiles; i++) {
    let direction = floor(random(4));

    if (direction === 0 && startX > 0) {
      startX--;
    } else if (direction === 1 && startX < mapSize - 1) {
      startX++;
    } else if (direction === 2 && startY > 0) {
      startY--;
    } else if (direction === 3 && startY < mapSize - 1) {
      startY++;
    }

    journeyPath.push(createVector(startX, startY));
  }
}

function generateRandomEncounters() {
  randomEncounters = [];

  let numEncounters = floor(random(mapSize / 2));

  for (let i = 0; i < numEncounters; i++) {
    let encounterX = floor(random(mapSize));
    let encounterY = floor(random(mapSize));

    randomEncounters.push(createVector(encounterX, encounterY));
  }
}
```

In this code, we use p5.js to create a canvas and draw the player map. The map consists of tiles, where the player can navigate using arrow keys. The

journey path, random encounters, and destination are randomly generated.

You can customize the map size, tile size, and tweak the generation logic to fit your game requirements. The code also includes basic event handling for encountering random events and reaching the destination.

Feel free to modify and enhance the code to add more features and game mechanics based on your RPG's needs.

Generating Mazes and Dungeons

To generate and visualize a maze with given width and length parameters, you can use a maze generation algorithm such as Recursive Backtracking or Prim's Algorithm.

Here's an example of how you can implement it using the Recursive Backtracking algorithm and the turtle module in Python:

```
# python - Maze Code 1

import random
import turtle

def generate_maze(width, height):
    # Initialize the maze grid with walls
    maze = [[1] * width for _ in range(height)]

    # Set the starting point
    start_x, start_y = random.randint(0, width - 1), random.randint(0,
height - 1)
    maze[start_y][start_x] = 0

    stack = [(start_x, start_y)]

    while stack:
        x, y = stack[-1]
        neighbors = []

        # Find unvisited neighbors
        if x > 1 and maze[y][x - 2]:
            neighbors.append((x - 2, y))
        if x < width - 2 and maze[y][x + 2]:
```

```
            neighbors.append((x + 2, y))
        if y > 1 and maze[y - 2][x]:
            neighbors.append((x, y - 2))
        if y < height - 2 and maze[y + 2][x]:
            neighbors.append((x, y + 2))

        if neighbors:
            next_x, next_y = random.choice(neighbors)
            maze[next_y][next_x] = 0
            maze[(y + next_y) // 2][(x + next_x) // 2] = 0
            stack.append((next_x, next_y))
        else:
            stack.pop()

    return maze

def visualize_maze(maze):
    turtle.speed(0)
    turtle.hideturtle()

    cell_size = 20
    turtle.penup()

    rows = len(maze)
    cols = len(maze[0])

    screen_width = cols * cell_size
    screen_height = rows * cell_size

    turtle.setup(screen_width + 50, screen_height + 50)
    turtle.setworldcoordinates(-20, -20, screen_width + 30,
screen_height + 30)

    for y in range(rows):
        for x in range(cols):
            if maze[y][x] == 1:
                turtle.goto(x * cell_size, y * cell_size)
                turtle.pendown()
                turtle.setheading(0)
                turtle.forward(cell_size)
                turtle.right(90)
                turtle.forward(cell_size)
                turtle.right(90)
                turtle.forward(cell_size)
                turtle.right(90)
                turtle.forward(cell_size)
                turtle.penup()

    turtle.exitonclick()

# Example usage:
```

```
width = int(input("Enter the width of the maze: "))
height = int(input("Enter the height of the maze: "))

maze = generate_maze(width, height)
visualize_maze(maze)
```

In the code above, the generate_maze() function implements the **Recursive Backtracking** algorithm to generate a maze. It initializes a grid of cells with walls, sets a starting point, and uses a stack to backtrack and carve paths until all cells are visited.

The visualize_maze() function uses the turtle module to visualize the generated maze. It sets up the turtle window based on the size of the maze and iterates through the grid, drawing walls where the value is 1.

You can input the desired width and height of the maze, and the code will generate and display the maze using the turtle graphics. You can click on the window to close it.

Need something a bit more browser based, here's an example of how you can generate and visualize a maze using the p5.js library in JavaScript:

```
let maze;
let cellSize = 20;

function setup() {
  createCanvas(800, 600);

  let width = floor(width / cellSize);
  let height = floor(height / cellSize);

  maze = generateMaze(width, height);
}

function draw() {
  background(255);

  for (let y = 0; y < maze.length; y++) {
    for (let x = 0; x < maze[y].length; x++) {
```

```
      if (maze[y][x] === 1) {
        let xPos = x * cellSize;
        let yPos = y * cellSize;

        stroke(0);
        fill(255);
        rect(xPos, yPos, cellSize, cellSize);
      }
    }
  }
}

function generateMaze(width, height) {
  let maze = [];

  // Initialize the maze grid with walls
  for (let y = 0; y < height; y++) {
    maze[y] = [];
    for (let x = 0; x < width; x++) {
      maze[y][x] = 1;
    }
  }

  // Set the starting point
  let startX = floor(random(width));
  let startY = floor(random(height));
  maze[startY][startX] = 0;

  let stack = [[startX, startY]];

  while (stack.length > 0) {
    let [x, y] = stack[stack.length - 1];
    let neighbors = [];

    // Find unvisited neighbors
    if (x > 1 && maze[y][x - 2]) {
      neighbors.push([x - 2, y]);
    }
    if (x < width - 2 && maze[y][x + 2]) {
      neighbors.push([x + 2, y]);
    }
    if (y > 1 && maze[y - 2][x]) {
      neighbors.push([x, y - 2]);
    }
    if (y < height - 2 && maze[y + 2][x]) {
      neighbors.push([x, y + 2]);
    }

    if (neighbors.length > 0) {
      let randomIndex = floor(random(neighbors.length));
      let [nextX, nextY] = neighbors[randomIndex];
```

```
        maze[nextY][nextX] = 0;
        maze[(y + nextY) / 2][(x + nextX) / 2] = 0;
        stack.push([nextX, nextY]);
      } else {
        stack.pop();
      }
    }

    return maze;
  }
```

To use this code, you'll need to include the p5.js library in your HTML file. You can create an HTML file with the following structure:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Maze Generator</title>
  https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.4.0/p5.js
  http://sketch1.js
  <style>body {padding: 0; margin: 0;} canvas {display: block;} </style>
</head>
<body>
</body>
</html>
```

Save the JavaScript code in a file named "sketch1.js" in the same directory as your HTML file.

When you open the HTML file in a web browser, it will display a maze generated using the Recursive Backtracking algorithm.

Here's an example of how you can generate and visualize a maze using **Prim's Algorithm** and the p5.js library in JavaScript:

```
let maze;
let cellSize = 20;

function setup() {
  createCanvas(800, 600);

  let width = floor(width / cellSize);
  let height = floor(height / cellSize);

  maze = generateMaze(width, height);
}

function draw() {
  background(255);

  for (let y = 0; y < maze.length; y++) {
    for (let x = 0; x < maze[y].length; x++) {
      if (maze[y][x] === 1) {
        let xPos = x * cellSize;
        let yPos = y * cellSize;

        stroke(0);
        fill(255);
        rect(xPos, yPos, cellSize, cellSize);
      }
    }
  }
}

function generateMaze(width, height) {
  let maze = [];

  // Initialize the maze grid with walls
  for (let y = 0; y < height; y++) {
    maze[y] = [];
    for (let x = 0; x < width; x++) {
      maze[y][x] = 1;
    }
  }

  // Set the starting point
  let startX = floor(random(width));
  let startY = floor(random(height));
  maze[startY][startX] = 0;

  let walls = [];
  addWalls(startX, startY);
```

```
    while (walls.length > 0) {
      let randomIndex = floor(random(walls.length));
      let [x, y] = walls[randomIndex];
      let neighbors = [];

      // Find visited neighbors
      if (x > 1 && maze[y][x - 2] === 0) {
        neighbors.push([x - 2, y, x - 1, y]);
      }
      if (x < width - 2 && maze[y][x + 2] === 0) {
        neighbors.push([x + 2, y, x + 1, y]);
      }
      if (y > 1 && maze[y - 2][x] === 0) {
        neighbors.push([x, y - 2, x, y - 1]);
      }
      if (y < height - 2 && maze[y + 2][x] === 0) {
        neighbors.push([x, y + 2, x, y + 1]);
      }

      if (neighbors.length === 1) {
        let [nx, ny, mx, my] = neighbors[0];
        maze[ny][nx] = 0;
        maze[my][mx] = 0;
        addWalls(x, y);
      }

      walls.splice(randomIndex, 1);
    }

    return maze;
  }

  function addWalls(x, y) {
    if (x > 1) walls.push([x - 2, y]);
    if (x < width - 2) walls.push([x + 2, y]);
    if (y > 1) walls.push([x, y - 2]);
    if (y < height - 2) walls.push([x, y + 2]);
  }
```

Make sure to include the p5.js library in your HTML file as shown in the previous example. Save the JavaScript code in a file named “sketch2.js” in the same directory as your HTML file.

When you open the HTML file in a web browser, it will display a maze generated using Prim’s Algorithm.

Need a bit more complexity, here is a visualisation of a grid-based dungeon with corridors, rooms, doors, and aspects of a maze using the p5.js library in JavaScript:

```
let dungeon;

let cellSize = 20;
let widthInCells;
let heightInCells;

function setup() {
  createCanvas(800, 600);

  widthInCells = floor(width / cellSize);
  heightInCells = floor(height / cellSize);

  dungeon = generateDungeon(widthInCells, heightInCells);
}

function draw() {
  background(255);

  for (let y = 0; y < dungeon.length; y++) {
    for (let x = 0; x < dungeon[y].length; x++) {
      let xPos = x * cellSize;
      let yPos = y * cellSize;

      if (dungeon[y][x] === "wall") {
        fill(0);
        rect(xPos, yPos, cellSize, cellSize);
      } else if (dungeon[y][x] === "corridor") {
        fill(255);
        rect(xPos, yPos, cellSize, cellSize);
      } else if (dungeon[y][x] === "room") {
        fill(200);
        rect(xPos, yPos, cellSize, cellSize);
      } else if (dungeon[y][x] === "door") {
        fill(255, 0, 0);
        rect(xPos, yPos, cellSize, cellSize);
      } else if (dungeon[y][x] === "entrance") {
        fill(0, 255, 0);
        rect(xPos, yPos, cellSize, cellSize);
      }
    }
  }
}
```

```
function generateDungeon(width, height) {
  let dungeon = [];

  for (let y = 0; y < height; y++) {
    dungeon[y] = [];
    for (let x = 0; x < width; x++) {
      dungeon[y][x] = "wall";
    }
  }

  let startX = floor(random(1, width - 1));
  let startY = floor(random(1, height - 1));
  dungeon[startY][startX] = "entrance";

  generateRooms(dungeon);
  generateCorridors(dungeon);
  generateDoors(dungeon);

  return dungeon;
}

function generateRooms(dungeon) {
  let numRooms = floor(random(5, 10));

  for (let i = 0; i < numRooms; i++) {
    let roomWidth = floor(random(3, 8));
    let roomHeight = floor(random(3, 8));
    let roomX = floor(random(1, widthInCells - roomWidth - 1));
    let roomY = floor(random(1, heightInCells - roomHeight - 1));

    for (let y = roomY; y < roomY + roomHeight; y++) {
      for (let x = roomX; x < roomX + roomWidth; x++) {
        dungeon[y][x] = "room";
      }
    }
  }
}

function generateCorridors(dungeon) {
  let startX = -1;
  let startY = -1;

  for (let y = 1; y < heightInCells; y += 2) {
    for (let x = 1; x < widthInCells; x += 2) {
      if (dungeon[y][x] === "room") {
        if (startX === -1) {
          startX = x;
          startY = y;
        } else {
          let currentX = startX;
          let currentY = startY;
```

```
  while (currentX !== x || currentY !== y) {
              if (currentX < x) {
                currentX++;
              } else if (currentX > x) {
                currentX--;
              } else if (currentY < y) {
                currentY++;
              } else if (currentY > y) {
                currentY--;
              }

              dungeon[currentY][currentX] = "corridor";
            }

            startX = -1;
            startY = -1;
          }
        }
      }
    }

  function generateDoors(dungeon) {
    for (let y = 1; y < heightInCells - 1; y++) {
      for (let x = 1; x < widthInCells - 1; x++) {
        if (dungeon[y][x] === "wall") {
          let isAdjacentToCorridor = false;

          if (
            dungeon[y - 1][x] === "corridor" ||
            dungeon[y + 1][x] === "corridor" ||
            dungeon[y][x - 1] === "corridor" ||
            dungeon[y][x + 1] === "corridor"
          ) {
            isAdjacentToCorridor = true;
          }

          if (isAdjacentToCorridor) {
            dungeon[y][x] = "door";
          }
        }
      }
    }
  }
```

Save the updated JavaScript code in a file named “sketch3.js” and make sure to include the p5.js library in your HTML file as shown in the previous examples. When you open the HTML file in a web browser, it will display a visual representation of a grid-based dungeon with corridors, rooms, doors, and an entrance.

Monsters

Here’s an example code for an OSR like monster generator that randomly selects a monster with typical stats, hit points, weapon, attitude, and their treasure:

```
# python - Monsters

import random

monsters = [
    {
        "name": "Goblin",
        "stats": {"AC": 13, "HP": "2d6", "Attack": "+4", "Damage":
"1d6"},
        "attitude": "Hostile",
        "treasure": "Copper coins"
    },
    {
        "name": "Orc",
        "stats": {"AC": 15, "HP": "2d8+2", "Attack": "+5", "Damage":
"1d8+2"},
        "attitude": "Hostile",
        "treasure": "Silver coins"
    },
    {
        "name": "Giant Spider",
        "stats": {"AC": 12, "HP": "3d8", "Attack": "+3", "Damage":
"1d6+1"},
        "attitude": "Aggressive",
        "treasure": "None"
    },
    # Add more monsters here...
]

def generate_monster():
    monster = random.choice(monsters)
```

```
        name = monster["name"]
        stats = monster["stats"]
        attitude = monster["attitude"]
        treasure = monster["treasure"]

        # Roll hit points
        hit_points = roll_dice(stats["HP"])

        # Generate the monster's description
        description = f"Monster: {name}\n"
        description += f"Attitude: {attitude}\n"
        description += f"Stats: {stats}\n"
        description += f"Hit Points: {hit_points}\n"
        description += f"Treasure: {treasure}\n"

        return description

    def roll_dice(dice):
        rolls, sides = map(int, dice.split("d"))
        return sum(random.randint(1, sides) for _ in range(rolls))

    # Generate a random monster
    monster_description = generate_monster()

    # Print the generated monster description
    print(monster_description)
```

In this code, we have a list called monsters containing dictionaries representing different monsters. Each monster has a name, stats (e.g., AC, HP, Attack, Damage), attitude, and treasure. You can add more monsters to the list with their respective attributes.

The generate_monster function selects a random monster from the list, rolls hit points based on the monster’s HP dice expression, and generates a description string including the monster’s name, attitude, stats, hit points, and treasure.

The roll_dice function is used to simulate rolling dice based on the provided dice notation (e.g., “2d6” for rolling two six-sided dice).

You can customize and expand upon this code by adding more monsters to the list, incorporating additional attributes, or modifying the output format

as per your requirements.

Names

Here's an example code that uses the "Random User Generator" API to generate random names for characters:

```
# python - ask randomuser.me for a name.

import requests

def generate_character_name():
    response = requests.get("https://randomuser.me/api/")
    if response.status_code == 200:
        data = response.json()
        name = data["results"][0]["name"]["first"]
        return name
    else:
        return None

# Generate a character name
character_name = generate_character_name()

# Print the generated character name
if character_name:
    print("Character Name:", character_name)
else:
    print("Failed to generate character name.")
```

In this code, we make a GET request to the "Random User Generator" API (https://randomuser.me/api/) to fetch a random user's data, which includes a first name. We extract the first name from the response data and return it as the generated character name.

The generated character name is then printed to the console.

Please note that APIs can evolve or change over time, so it's important to refer to the documentation of the chosen API for any specific requirements

or restrictions when using the “Random User Generator” API or any other similar name generation APIs.

https://github.com/RandomAPI/Randomuser.me-Node

Random Encounters

Here’s an example code for a random encounter generator that reads input from a formatted text file. The file syntax and format are as follows:

File Syntax:

- Each line in the file represents a unique encounter.
- The format for each line is as follows: <description>|<difficulty>|<location>|<reward>

File Format:

- <description>: A brief description of the encounter.
- <difficulty>: An integer representing the difficulty level of the encounter.
- <location>: The location where the encounter takes place.
- <reward>: A reward or treasure associated with the encounter.

Example File (encounters.txt):

```
Goblin ambush|2|Forest|10 gold coins
Mysterious cave|3|Mountains|Magical artifact
Bandit attack|4|Road|25 silver coins
Ancient ruins|5|Desert|Ancient treasure chest
```

Now, here’s the code to read the file and generate a random encounter:

```python
#python - Random Encounters read from a file

import random

def read_encounter_file(filename):
    encounters = []
    with open(filename, "r") as file:
        for line in file:
            line = line.strip()
            if line:
                encounter_data = line.split("|")
                if len(encounter_data) == 4:
                    encounter = {
                        "description": encounter_data[0],
                        "difficulty": int(encounter_data[1]),
                        "location": encounter_data[2],
                        "reward": encounter_data[3]
                    }
                    encounters.append(encounter)
    return encounters

def generate_random_encounter(encounters):
    if encounters:
        encounter = random.choice(encounters)
        return encounter
    else:
        return None

# Read encounters from the file
encounters = read_encounter_file("encounters.txt")

# Generate a random encounter
random_encounter = generate_random_encounter(encounters)

# Print the generated random encounter
if random_encounter:
    print("Random Encounter:")
    print("Description:", random_encounter["description"])
    print("Difficulty:", random_encounter["difficulty"])
    print("Location:", random_encounter["location"])
    print("Reward:", random_encounter["reward"])
else:
    print("No encounters available.")
```

In this code, the read_encounter_file function reads the encounter details from the specified file. It parses each line and creates a dictionary representing an encounter with the description, difficulty, location, and reward. The encounters are stored in a list.

The generate_random_encounter function randomly selects an encounter from the provided encounters list. If encounters are available, it returns a random encounter dictionary; otherwise, it returns None.

The encounters are read from the file using the read_encounter_file function, and a random encounter is generated using generate_random_encounter. Finally, the details of the random encounter are printed to the console.

You can modify the file syntax, format, and file name as per your requirements. Make sure the text file follows the specified syntax and format to ensure proper parsing and generation of random encounters.

There are APIs available that you can call to generate random encounters. Here are a few examples:

- **D&D 5th Edition API (D&D5eAPI):** The D&D5eAPI provides various endpoints to retrieve data related to Dungeons & Dragons 5th Edition. You can make use of the /monsters endpoint to fetch information about monsters, which can be used to generate random encounters. You can find more information about the API and its endpoints in the D&D5eAPI documentation.
- **Open5e API:** Open5e is an open-source API that provides data and resources for Dungeons & Dragons 5th Edition. It offers endpoints to access monster data, including their attributes, abilities, and more. You can refer to the Open5e API documentation to learn about the available endpoints and how to use them.
- **Roleplaying APIs (RPGAPIs):** RPGAPIs is a collection of APIs specifically designed for role-playing games. It includes various endpoints for generating random encounters, such as

/encounters/random, which provides a random encounter based on specified parameters. You can explore the RPGAPIs documentation to understand the available endpoints and how to integrate them into your code.

Before using any API, make sure to review their documentation, terms of use, and any usage limitations or requirements. Each API may have its own syntax and authentication process for making API calls.

Magic Items

Here’s an example code to generate a random magic item based on a list of common items, magic powers, effects, and their usage limits:

```
#python - magic items

import random

common_items = [
    "Ring",
    "Amulet",
    "Potion",
    "Scroll",
    "Wand",
    "Staff",
    "Bracelet",
    "Gem"
]

magic_powers = [
    "Fire",
    "Ice",
    "Teleportation",
    "Invisibility",
    "Healing",
    "Summoning",
    "Transformation",
    "Protection"
]

effects = [
    "Increase damage",
    "Grant temporary flight",
```

```
        "Grant night vision",
        "Create a force field",
        "Grant resistance to elements",
        "Cast a powerful spell",
        "Summon a creature",
        "Grant enhanced senses"
    ]

    def generate_magic_item():
        item = random.choice(common_items)
        power = random.choice(magic_powers)
        effect = random.choice(effects)
        uses = random.randint(1, 5)  # Random number of uses

        return f"{item} of {power}: {effect} ({uses} uses)"

    # Generate a random magic item
    magic_item = generate_magic_item()

    # Print the generated magic item
    print("Random Magic Item:")
    print(magic_item)
```

In this code, we have lists common_items, magic_powers, and effects that contain the respective options for generating a magic item. The generate_magic_item function selects a random item, power, effect, and a random number of uses between 1 and 5. It then combines these elements into a formatted string representing the magic item.

The usage limits are determined by the randomly chosen number of uses. You can adjust the range of the random number generation based on your preference or requirements.

The code ensures that simple low-power items are more common since they have an equal chance of being selected from their respective lists. If you want to adjust the probabilities or balance the distribution of items, you can modify the lists or introduce weights to the random selection process.

Feel free to customize the code by adding more options to the lists, expanding the effects, or enhancing the formatting of the generated magic item.

Resources

Here’s a list of online resources for writing code for RPGs.

1. **RPG Toolkit**

   Summary: RPG Toolkit is a comprehensive set of tools and resources for creating and running RPGs. It includes an editor for designing game worlds, a scripting language, and a game engine for implementing your RPG mechanics.

   Link: RPG Toolkit

2. **Roll20**

   Summary: Roll20 is a popular virtual tabletop platform that provides a wide range of tools for playing and creating RPGs online. It offers features like character sheets, dice rolling, map creation, and a marketplace for game assets.

   Link: Roll20

3. **RPG Maker**

   Summary: RPG Maker is a software that enables game developers to create their own RPGs without extensive coding knowledge. It offers a visual interface for designing maps, characters, and dialogues, along with a scripting system for customizing game mechanics.

   Link: RPG Maker

4. **Tiled**

   Summary: Tiled is a flexible map editor suitable for RPGs and other game genres. It allows you to design and construct tile-based maps with layers, objects, and custom properties. It supports various map formats and offers plugins for integration with game engines.

   Link: Tiled Map Editor

5. **Unity**

   Summary: Unity is a powerful game development engine that can be used to create a wide range of games, including RPGs. It provides a visual editor, scripting capabilities in C#, and a vast asset store

for acquiring RPG-related assets, scripts, and plugins.

Link: Unity

6. **Godot**

   Summary: Godot is an open-source game engine suitable for RPG development. It features a visual editor, a node-based scene system, and a scripting language (GDScript) for implementing game logic. It has an active community and extensive documentation.

   Link: Godot Engine

7. **GitHub**

   Summary: GitHub is a platform for version control and collaborative development. It provides a space for sharing and discovering open-source RPG projects, code samples, and libraries. You can explore repositories, contribute to existing projects, or start your own.

   Link: GitHub

These resources offer a range of tools, engines, editors, and communities to support the creation of RPGs. Depending on your specific needs and preferences, you can explore these resources to find the most suitable tools and platforms for your RPG development journey.

DriveThruRPG

DriveThruRPG is an online marketplace that specializes in digital and print-on-demand role-playing game (RPG) products. It offers a vast collection of RPG rulebooks, supplements, adventures, and resources from various publishers. It provides a convenient platform for both independent creators and established companies to distribute their RPG materials to a wide audience.

When it comes to solo play resources, DriveThruRPG offers a range of products designed specifically for solo role-playing experiences. These resources cater to players who prefer to engage in RPGs on their own, without the need for a traditional game master or a group of players. Solo play resources often provide guidance, rules, or scenarios tailored to

solo adventures, enabling players to enjoy immersive storytelling and challenging gameplay even when playing alone.

Here are some popular solo play resources available on DriveThruRPG:

- “Ironsworn” by Shawn Tomkin: It’s a complete RPG system designed for solo and cooperative play. It features a dark fantasy setting and provides a unique system for resolving actions and tracking progress.
- “Mythic Game Master Emulator” by Word Mill: This resource offers a set of tools and guidelines for solo role-playing. It helps simulate the decision-making and improvisation aspects of a game master, allowing players to create engaging stories and encounter unexpected events.
- “Scarlet Heroes” by Kevin Crawford: It’s a retro-style fantasy RPG tailored for solo play or small groups. It includes rules for solo adventuring, scalable encounters, and guidelines for running NPCs.
- “The Solo Adventurer’s Toolbox” by Paul Bimler: This resource provides a collection of solo play techniques, tables, and tools to enhance solo role-playing experiences. It offers prompts for generating plots, encounters, and exploring various genres.
- “Four Against Darkness” by Ganesha Games: It’s a solo dungeon-crawling game where players control a party of four adventurers. It provides random dungeon generation, encounters, and character progression mechanics for solo play.

These are just a few examples of the many solo play resources available on DriveThruRPG. You can explore the site further to find a wide range of rulebooks, supplements, adventures, and tools specifically designed for solo play in different RPG genres and systems.

 Code, Code for Games, GM, python, RPG, solo play

