5.8.5 Level Design

O jogo vai se passar somente em um cenário, pois se trata de um ambiente de cativeiro da minhoca, que é característica básica da vermicompostagem. Tem a possibilidade desse cativeiro mudar de cor, conforme a qualidade do adubo.

5.8.6 Arte

A arte é baseada no minhocário da UTFPR, tendo apenas como plano de fundo a mudança de plataformas, conforme a fase. Foi escolhida o estilo *cartoon* para dar maior diversão ao jogador, pois não se sente na obrigação de estar em um simulador ou um jogo de primeira pessoa, por exemplo. Outro fator para escolha é pela habilidade artística adquirida com *cartoons* ao longo dos anos.

Com esse estilo de arte, foi possível expandir o design do jogo, transferindo para o papel, no formato de colorir como mostrado através da Figura 12. Essa expansão foi fundamental para ensinar as crianças do ensino fundamental durante as atividades sobre o tema.

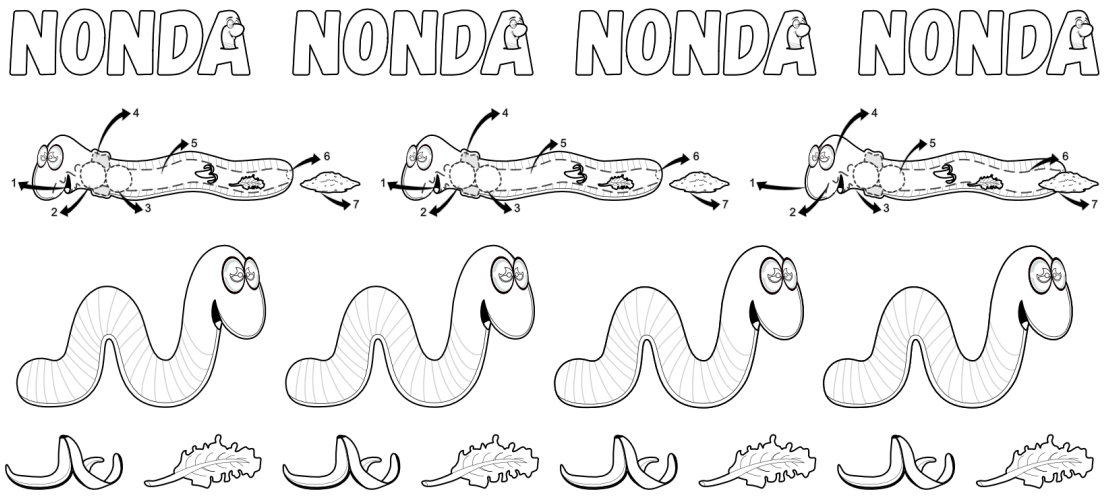

Figura 12 - Desenho para Colorir

6 DESENVOLVIMENTO

Até o capítulo 5, este trabalho concentrou-se em mostrar a fundamentação teórica seguida do ANEXO A – GAME DESIGN DOCUMENT DO JOGO NONDA para entender aspectos de criação do jogo Nonda, as ferramentas que faz parte do processo de desenvolvimento e os métodos a serem utilizados para atingir o objetivo. Nesse capítulo será abordado o processo de construção, incluindo diagramas, trechos de códigos e considerações.

Para entender o diagrama de sequência é necessário relembrar o *storyboard* do jogo Nonda (Figura 5).

6.1 Diagrama de Sequência

No contexto do jogo Nonda, o diagrama de sequência narra o fluxo do jogo, deixando claro quais cenas necessitarão de quais elementos e em qual ordem. Na Figura 13, primeiro é mostrado ao usuário a tela principal do jogo, seguido da tela de tutorial (que se repete a cada fase). Logo após surge a tela do jogo para o usuário interagir, pontuar e avançar de fase. É importante mencionar que existem telas que podem ser acessadas durante o jogo como o Tutorial ou o Menu de Opções. Seguindo o fluxo do jogo, a próxima tela informa a performance do jogador na fase específica (também conhecido como *End Screen*) que dá acesso a uma próxima fase, jogar novamente a fase, caso não pontuar suficiente ou ao menu principal. Essa sequência do storyboard faz o *game design* fluir organicamente.

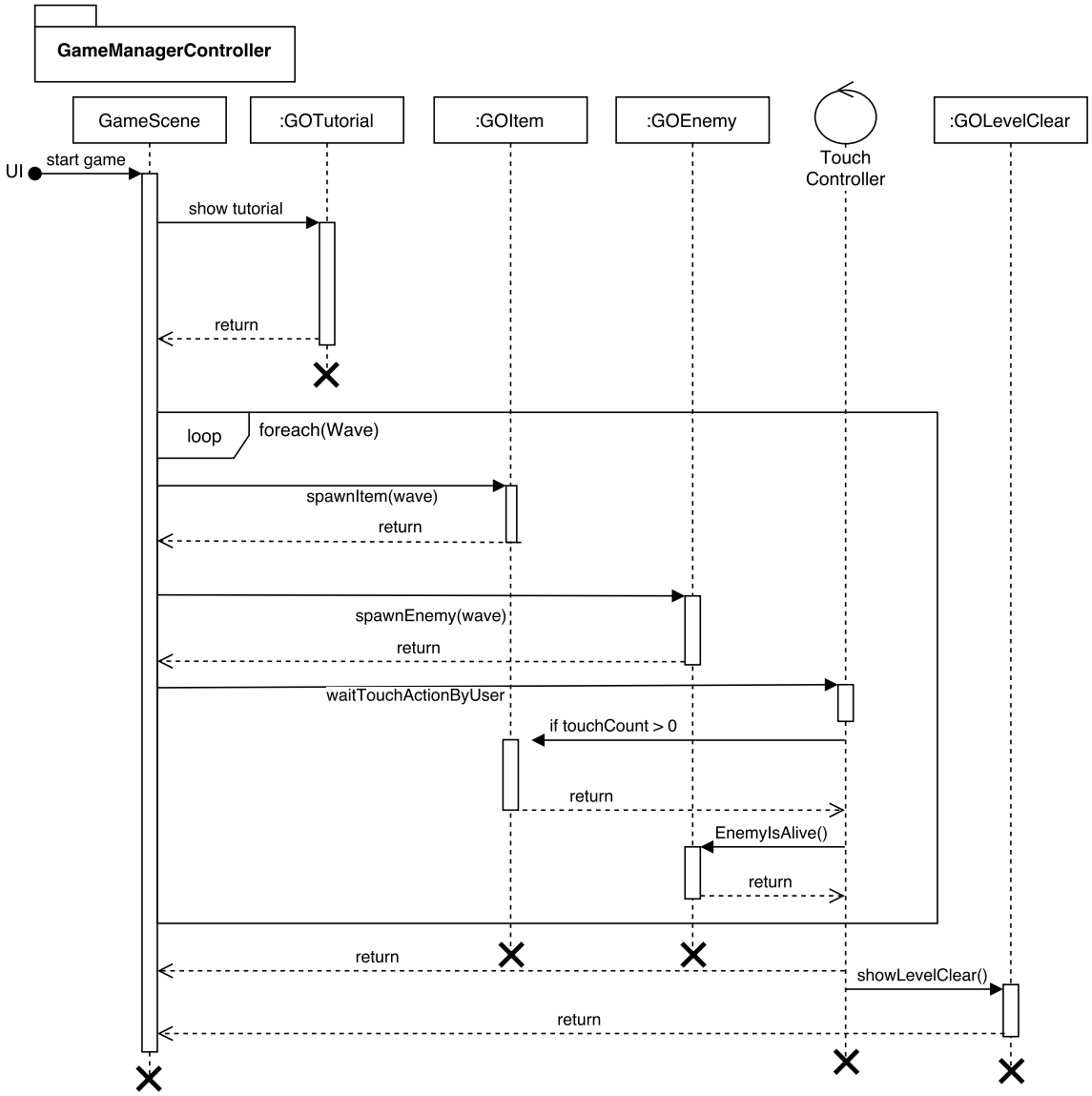

Figura 13 - Diagrama de Sequência

6.2 Diagrama de Classe

O diagrama de classe foi modelado seguindo o conceito da arquitetura Model View Controller descrito na seção 5.2 separando a modelagem de negócio da lógica de programação e da interface de usuário. O diagrama é composto de 21 classes e as três principais são Character, Level Manager e Item Controller; as demais classes tem associações e herança com essas classes como mostrado na . Pode ser observado que a Classe *Character* é a base tanto para implementar o jogador quanto para o predador. Outra classe importante a citar é o *Spawner Controller*, que foi implementada para dar base às outras como spawnar[2] itens coletados e *spawnar* predadores.

[2] *Spawn* – Termo que descreve o nascimento de um jogador, inimigo item, etc. Respawn é o termo usado para um jogador voltar à vida dentro do jogo.

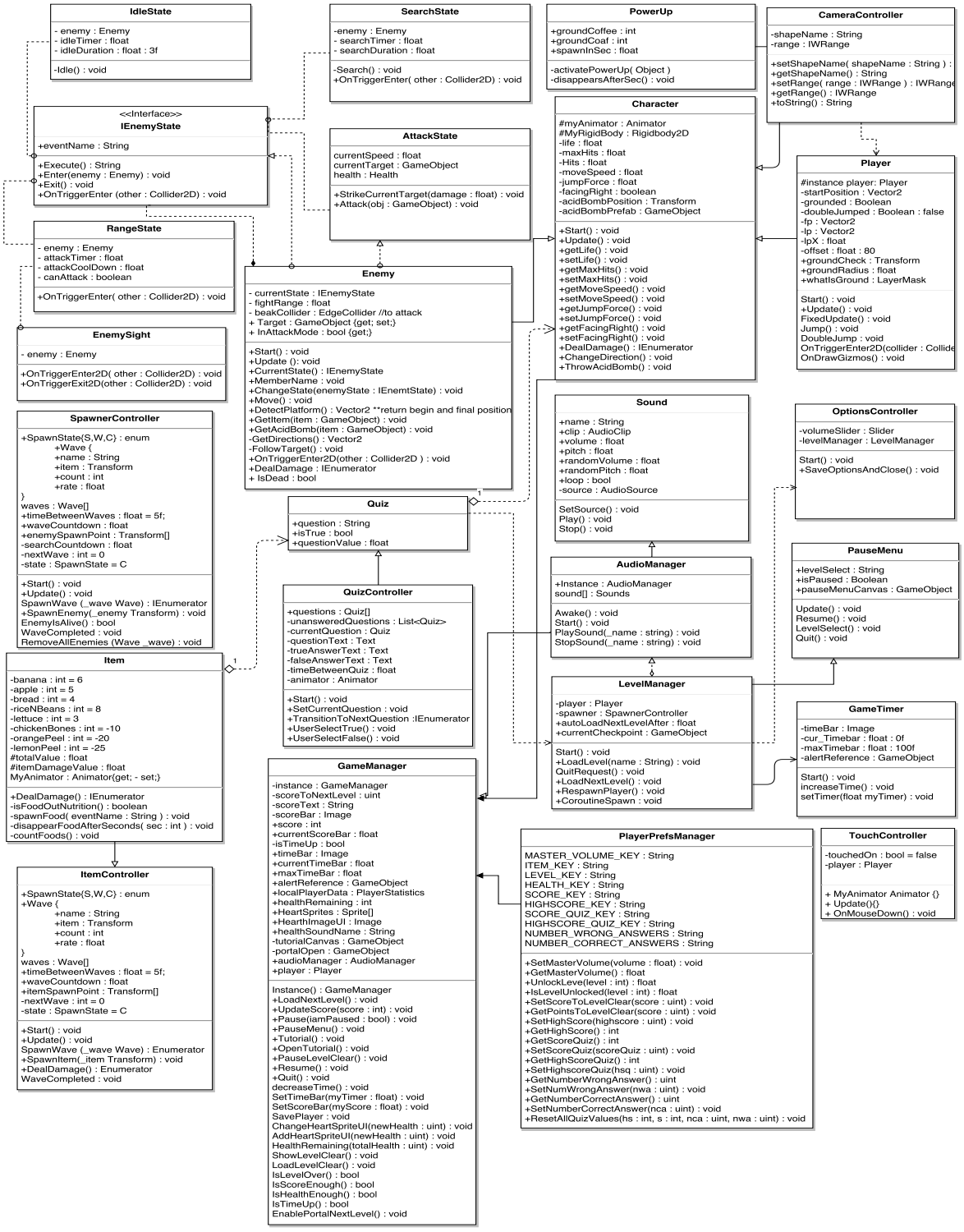

Figura 14 – Diagrama de Classe seguindo o conceito MVC

6.3 Fluxograma de Animação

A personagem Nonda possui diversos estados de animação, de acordo com um comportamento gerado pelo *gameplay* ou pela intervenção do usuário através do toque na tela. São atribuídos cinco estados diferentes que Nonda: Andar, Correr, Pular, Cair, Pousar. Esses estados de animação são definidos abaixo, na Figura 15. Para mudar de estado, algum outro estado precisa intervir em tempo de execução do jogo, no método *Update()*, que é chamado a cada *frame*[3]. O estado de animação padrão da Nonda é Andar. Isso significa que sempre a velocidade é maior do que 0 e, se o jogador executar ações do tipo pular ou mudar de direção, implicará mudança de estado de animação resultando em conjunto de sprites totalmente diferente para que represente tal estado de animação.

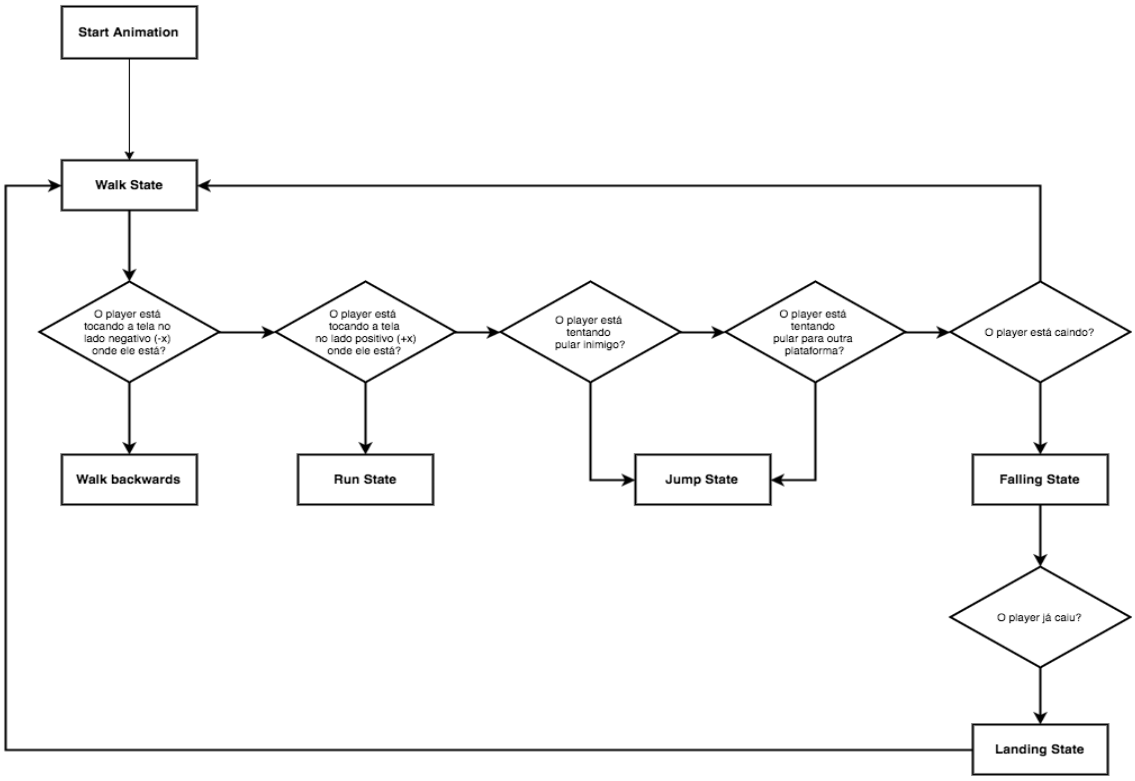

Figura 15 – Fluxograma Animação do personagem Nonda (Player)

[3] *Frame*: É a capacidade máxima que o processador consegue reproduzir imagens dentro de 1 segundo. A essa denominação, chama se *frames per second (fps)* ou em português frames por segundo.

6.4 Codificação

No presente capítulo, foca-se na prática de desenvolvimento, usando a linguagem C#, tendo subtópicos para códigos que implementaram função descritas no Diagrama de Classe usando o padrão de projeto MVC.

6.4.1 Coroutines

Coroutine (Co-rotina, em português) é uma função da própria *game engine* que fornece execução de uma tarefa sequencial (sem divisão de processador). Coroutine funciona da seguinte forma: a função salva o último ponto que parou, e no próximo frame executa o restante do código. O ganho em relação às *threads* (*threads* ou *multithreads* são funções que usam sincronia de várias funções distintas para executar animações, colisões, entradas fornecidas pelo usuário, entre outros) é que não exige alto processamento computacional, especialmente em caso de dispositivo móvel com baixa memória. O Unity 3D usa a interface IEnumerator para implementar Coroutine, isso significa que une duas interfaces (IEnumerator e IEnumerable) para que possa numerar, através de um cursor interno para o índice atual e fazer o gerenciamento para o ponto real eliminando verificações que espendem memória. É usado a declaração *yield* como marcador para que continue após essa declaração na próxima vez que for chamado. Semelhantemente ocorre com a declaração *return*. Abaixo, na Figura 16 pode-se observar o código extraído da classe ItemController.cs que tem adiciona itens na tela em forma de *waves*[4]. No método *SpawnWave*(Wave *wave)* dentro do um laço de repetição for encontra-se uma chamada para o método *SpawnItem(_wave.item*) que adiciona na jogo a quantidade de items definidos pelo tamanho do laço de repetição. Logo após, é chamada a declaração *yield* que retorna uma função *WaitForSeconds(1f/rate),* esperando um tempo passado entre parâmetros para dar continuidade na execução do método. Logo após sair do

[4] Waves - Termo usado para caracterizar um grupo de inimigos, itens, power-ups que vem em um certo tempo em uma quantidade pré-determinada durante o gameplay de um jogo.

laço, o Enum muda de estado da função para *Waiting* e passa a executar o que vem após a declaração *yield*.

```
void Update(){
        if(state == SpawnState.WAITING){
                if(!ItemHasEnergy()){
                        WaveCompleted();
                }else{
                        return;
                }
        }
        if(waveCountdown <= 0){
                if (state != SpawnState.SPAWNING){
                        HandController.Instance.PlayHandAnimation();
                        StartCoroutine( SpawnWave( waves[nextWave] ) );
                }
                else{
                        waveCountdown -= Time.deltaTime;
                }
        }
{
bool ItemHasEnergy(){
        searchCountdown -= Time.deltaTime;
        if(searchCountdown <= 0f){
                searchCountdown = 1f;
                if(GameObject.FindGameObjectWithTag ("Item") == null){
                        return false;
                }
        }
        return true;
}

IEnumerator SpawnWave(Wave _wave){
        state = SpawnState.SPAWNING;
        for (int i = 0; i < _wave.count; i++){
                SpawnItem(_wave.item);
                yield return new WaitForSeconds(1f/_wave.rate);
        }
        state = SpawnState.WAITING;
        yield break;
}

public void SpawnItem(Transform _item, Transform _touchColorItem){
        Transform _spawnPoint = itemSpawnPoint[ Random.Range (0,
itemSpawnPoint.Length) ];
        Instantiate(_item, _spawnPoint.position, _spawnPoint.rotation);
}

        public void SpawnItem(Transform _item){
                Transform _spawnPoint = itemSpawnPoint[ Random.Range
(0,itemSpawnPoint.Length) ];
                Instantiate(_item, _spawnPoint.position,
_spawnPoint.rotation);
        }
```

Figura 16 - Código usando Coroutines

Com uso de coroutines, foi possível seguir o diagrama de classe e conectar classes, solidificou dependências apenas criando métodos específicos. Outra contribuição importante

foi a capacidade de eliminar processamento de memória (dito anteriormente) com a habilidade de pausar a execução e retornar o controle para o Unity, e quando volta a execução continua a partir do ponto onde foi deixado.

Coroutines foi a base para implementar todas classes que envolve criação de objetos em tempo de jogo como as classes *ItemController.cs*, *SpawnerController.cs* que representa os Itens e Predadores, respectivamente. Porém, existe uma classe que é exceção: ItemSpawner.cs. Essa classe deposita os itens não saudáveis em tempos aleatórios e faz desaparecer independente se todos foi tocado pelo jogador ou não. A classe traz um conceito simples de inteligência artificial para jogo, embora o desenvolvimento não tem foco nessa área. Na Figura 17, o trecho do código da classe em questão verifica se é hora de spawnar novos objetos na cena, caso seja positivo é chamado o método *isTimeToSpawn()*. Esse método verifica se existe algum item na cena e se ainda pode ser depositado item. Caso seja positivo, é executado o método *SpawnWave()* com parâmetros necessários.

```
void Update (){
     if (waveCountdown <= 0) {
          isTimeToSpawn ();
     } else {
          waveCountdown -= Time.deltaTime;
     }
}

void isTimeToSpawn (){
     waveCountdown = timeBetweenWaves;
     if (GameObject.FindGameObjectWithTag ("ItemN") == null) {
          if (nextWave > waves.Length - 1) {
               nextWave = 0;
               Debug.Log ("All bad waves completed! Looping...");
               return;
          } else {
               SpawnWave (waves [nextWave]);
               nextWave++;
          }
     }
}
```

Figura 17 - Trecho de código da classe ItemSpawner