Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Victor Domiciano Mendonça

Inteligência Artificial em um jogo procedural do gênero *Tower Defense*

São Paulo
Dezembro de 2018

Inteligência Artificial em um jogo procedural do gênero *Tower Defense*

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado

Supervisor: Profº Dr. Flávio Soares Corrêa da Silva

São Paulo
Dezembro de 2018

Resumo

Neste trabalho, foi desenvolvido o jogo *Path of The Wicked,* um jogo do gênero *Tower Defense*, cujo mapa é gerado proceduralmente. *Tower Defense* é um gênero de jogos de estratégia, em tempo real ou em turnos, com foco em aquisição e aplicação de recursos e posicionamento de unidades de proteção em um mapa. *Path of The Wicked* foi desenvolvido utilizando a *Godot*, uma *game engine* multiplataforma *open source*, na linguagem *GDScript*, uma linguagem de programação dinâmica de alto nível. O terreno do jogo é composto de um *grid* hexagonal, mapeado em um grafo ponderado dirigido, e a inteligência artificial se resume em algoritmos de busca nesse grafo, como o algoritmo A*.

Palavras-chave: desenvolvimento de jogos, *tower defense*, geração procedural, inteligência artificial, *Godot Engine*.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação e objetivo

A área de desenvolvimento de jogos eletrônicos apresenta diversos campos de pesquisa relacionados à Ciência da Computação, notavelmente a computação gráfica, gerando modelos e animações tridimensionais excepcionalmente realistas, a inteligência artificial, criando agentes inteligentes capazes de tomar decisões que emulam comportamentos humanos, e a geração procedural, produzindo dados e elementos de naturezas variadas a partir de algoritmos meticulosos [Yannakakis, 2018]. Especificamente sobre inteligência artificial, a produção de jogos pode ter um foco diferente das aplicações reais, pois nem sempre o ideal é realizar as ações ótimas.

Nesse sentido, a literatura relacionada à *game design* expressa de maneira pertinente: é mais interessante que os agentes presentes em um jogo atuem com ênfase no entretenimento e nos desafios que proporcionam ao jogador do que agir utilizando a melhor abordagem possível contra humanos [Short, 2017]. Um dos desafios que surge durante o desenvolvimento de um jogo é a implementação de algoritmos eficientes, pois a execução é feita em tempo real, ou seja, todo o processamento de uma etapa lógica, renderizado em um quadro, deve ser feito em menos de 16 milissegundos.

O USPGameDev é o grupo de estudos sobre desenvolvimento de jogos analógicos e eletrônicos da USP, e a motivação deste trabalho é em decorrência da minha participação no grupo desde 2015, quando ingressei no Bacharelado em Ciência da Computação do Instituto de Matemática e Estatística da Universidade de São Paulo. O objetivo é aplicar conhecimentos que adquiri durante a graduação, relacionados aos algoritmos e estruturas de dados, e durante as atividades extracurriculares que participei no USPGameDev, desenvolvendo jogos, aprendendo conceitos de *game design* e adquirindo experiência na utilização da *Godot Engine*.

Capítulo 2

Conceitos e Ferramentas

2.1 O gênero *Tower Defense*

Jogos do gênero *Tower Defense* são jogos de estratégia, em tempo real ou em turnos, com foco em aquisição e aplicação de recursos e posicionamento de unidades de proteção em um mapa. Na sua forma mais simples, jogos de *Tower Defense* consistem de um jogador comprando e organizando torres defensivas que atiram em diferentes tipos de inimigos, chamados *creeps*. Para cada *creep* destruído pelas torres, o jogador ganha recursos correspondentes ao nível de dificuldade desse *creep*. A interação com as torres baseia-se na compra, colocação e aperfeiçoamento dessas unidades, que atiram automaticamente nos *creeps* que se aproximam.

O jogo é organizado em ondas (*waves*), em que cada onda é composta por um conjunto de *creeps* que surgem em um ponto do mapa e rumam em direção à base. Os inimigos que alcançam a base geram uma punição ao jogador, geralmente através da diminuição dos pontos de resistência da base ou da subtração de recursos [Schell, 2015]. Se os pontos de resistência, ou os recursos do jogador chegarem à zero, o jogador perde, e diz-se que sobreviveu ao número de ondas correspondente à onda atual menos uma. O jogo pode ser organizado em etapas, cada etapa com um número determinado de ondas, e o jogador só desbloqueia a próxima etapa após concluir a atual.

Os elementos que a maioria dos jogos de *Tower Defense* apresentam são os seguintes [Avery, 2011]:

2.1.1 Terreno

O mapa de um *Tower Defense* define como são os caminhos pelos quais os *creeps* irão se locomover, e quais são as áreas ao redor em que é permitido colocar torres. A complexidade de um terreno depende da quantidade de caminhos, da sinuosidade deles, do comprimento, se há ramificações ou não, entre outros. Um terreno pode restringir a área ao redor dos caminhos colocando, por exemplo, um lago e definindo que torres apenas podem ser colocadas em terra. É possível também que uma partida se inicie com um conjunto de

torres previamente estabelecidas no terreno, e o jogador poderá adicionar mais torres posteriormente.

É dito que um terreno tem baixa dificuldade para o jogador se existem poucos caminhos e várias áreas de impacto, e que tem alta dificuldade se existem vários caminhos com ramificações e poucas áreas de impacto. Uma área de impacto é uma região do mapa em que os *creeps* permanecem na área de alcance de uma torre por um longo período, aumentando a eficácia dessa torre. Essa característica torna bastante vantajoso para o jogador colocar uma torre em uma área de impacto, e geralmente é associada às estratégias vencedoras.

Figura 2.1: Uma torre em uma área de grande impacto

2.1.2 Torres

A construção de torres requer estratégia e captação de recursos. A maioria dos jogos disponibilizam diferentes tipos de torres com diferentes capacidades e preços. Em alguns jogos clássicos, algumas torres são capazes apenas de atingir *creeps* específicos, por exemplo torres que atiram apenas em *creeps* aéreos e torres que atiram apenas em *creeps* terrestres. Geralmente torres que atingem tanto *creeps* aéreos quanto terrestres são mais fracas.

Alguns atributos das torres são: velocidade do projétil, taxa de disparo, alcance, dano que causa ao atingir um *creep* e efeitos. Esses atributos podem ser melhorados através da compra de torres mais avançadas, ou do aperfeiçoamento de torres já adquiridas.

São efeitos comuns encontrados em jogos de *Tower Defense*: disparos com múltiplos projéteis, projéteis que causam dano em vários *creeps* simultaneamente, que diminuem a velocidade do *creep* ou o paralisam por um determinado período.

2.1.3 *Creeps*

Os tipos de *creeps* encontrados nos jogos são influenciados pelos tipos de torres presentes, e devem ser cuidadosamente balanceados. Geralmente os *creeps* correspondem à tipos compatíveis, como aéreos ou terrestres, gelo ou fogo, etc. Os atributos de um *creep* podem ser resumidos em: pontos de vida, velocidade e resistência (ou armadura). *Creeps* que se locomovem mais devagar comumente tem pontos de vida maiores, e os mais rápidos requerem menos projéteis para destruí-los. Quanto mais difícil for destruir um *creep*, maior o número de recursos recompensado ao jogador.

Vários tipos de jogos também possuem um tipo especial de *creep,* geralmente único, apelidado de "chefe", que é particularmente difícil de destruir. Esses tipos especiais de *creeps* costumam aparecer ao final das etapas, e podem ter uma onda reservada exclusivamente para ele. Pode haver um tipo de regra especial que estabelece que se um "chefe" atinge a base, o jogador perde.

2.1.4 Estratégias

As estratégias de um *Tower Defense* podem variar de acordo com os elementos presentes no jogo. Diferentes tipos de *creeps* requerem diferentes estratégias, e um jogo com bom *game design* deve ser balanceado para ser desafiador. No entanto, as estratégias podem ser agrupadas na resolução de dois problemas: gerenciamento de recursos e posicionamento de unidades.

Na maioria dos casos, gerenciamento de recursos envolve escolher entre gastar os pontos em grandes quantidades de unidades baratas, ou salvar os pontos para adquirir posteriormente unidades mais caras e mais eficientes. Em geral, a melhor estratégia é comprar o menor número possível de unidades baratas, o suficiente para não sofrer as punições geradas pelos *creep*s quando atingem a base, e guardar os recursos para as unidades mais eficientes.

O posicionamento das unidades depende completamente do mapa e dos *creeps* contidos no jogo. Pode ser útil manter uma boa linha de frente para conter os *creeps* assim que eles surgem no mapa, mas *creeps* mais rápidos podem conseguir passar por essa proteção. Dessa forma, seriam necessárias unidades adicionais no restante do mapa. Em jogos nos quais os projéteis das torres possuem efeitos, também é interessante concentrar torres em uma mesma área, se possível de grande impacto, pois os efeitos contribuem em uma eliminação mais rápida dos *creeps*.

2.2 Jogos eletrônicos

No desenvolvimento de jogos eletrônicos, o código que compõe o *software* possui alguns padrões peculiares e características importantes que devem ser levadas em consideração no processo de criação. Antigamente, o desenvolvimento e a execução dos programas seguia um modelo que era composto por algumas etapas: o programador escrevia o código, executava, e esperava por uma saída com os resultados. Esses tipos de programas são conhecidos como programas "*batch mode*". Assim que o programa terminava o processamento, ele encerrava. Existem vários tipos de programas que ainda são assim atualmente.

Com a criação de novos paradigmas, surgiram os programas interativos. Nas primeiras versões, os programas executavam um bloco de código e aguardavam a entrada do usuário. Em jogos, a execução é semelhante à essa, com a diferença de que o programa não fica travado se o usuário não fornece nenhuma entrada.

```
1    while (true) {
2        processInput();
3        update();
4        render();
5    }
```

Esse *loop* é conhecido como *game loop* [Nystrom, 2014]. De maneira resumida, o jogo processa o *input* do jogador, realiza cálculos para atualizar o estado do jogo, e renderiza o resultado na tela. É importante também destacar que a quantidade de vezes que este bloco de código é executado por segundo influencia diretamente na velocidade lógica do jogo. Por essa razão, o código deve ser eficiente, e idealmente esse *loop* deve ser executado 60 vezes por segundo, em intervalos de tempo iguais.

Para auxiliar na produção de jogos, existem *frameworks* especializados que possuem um ambiente próprio de desenvolvimento e que disponibilizam diversas ferramentas que são muito comuns nesse processo. Esses tipos de *framework,* chamados de *Game Engines*, têm suporte para renderizar gráficos em 2D e 3D, lidar com simulações físicas, inteligência artificial, sons, e a maioria dos elementos presentes em jogos.

2.3 *Godot Engine*

Godot Engine é uma *game engine* multiplataforma usada na criação de jogos 2D e 3D. É um *software open source* sob a licença MIT*,* desenvolvido de forma independente pela comunidade.

Figura 2.2: Logo da *Godot Engine*

Para utilizar essa *engine,* é importante entender alguns conceitos básicos.

2.3.1 Nó

Existem muitos tipos diferentes de nós, cada um com seus atributos e objetivos. Por exemplo, um nó pode mostrar uma imagem, tocar um som, ou exibir um modelo 3D. No entanto, todos os nós têm as seguintes propriedades:

- Tem um nome
- Tem atributos que podem ser editados
- Pode receber um *callback* de execução todos os *frames*
- Pode ser estendido (para receber mais funções)
- Pode ser adicionado à outro nó como filho

O último atributo tem uma consequência notável: Nós podem ter outros nós como filhos, então gerando uma estrutura de árvore.

Na Figura 2.3, o nó raiz da árvore é o nó HUD. Os nós *NextWave*, *Panel*, *Gold*, *Popup*, *BlurShader* e *Notifications* são filhos do nó HUD, e os nós *PanelArea*, *ShopButton*, *Inventory*, *Items* e *Discard* são filhos do nó *Panel*. Outros nós também estão presentes na árvore, mas estão ocultos.

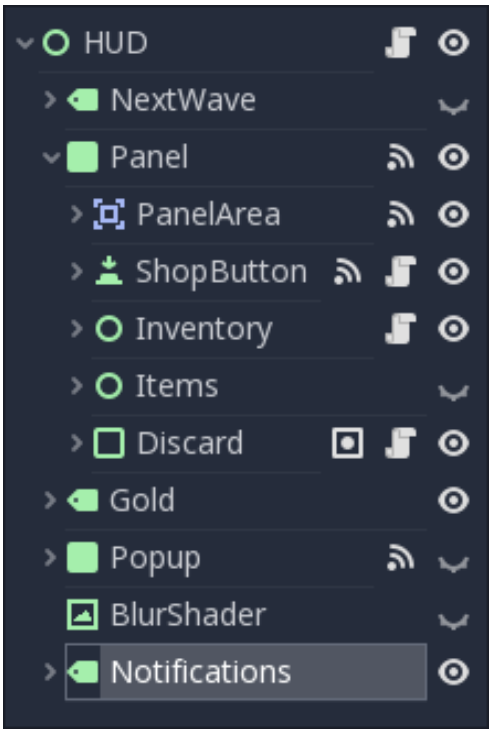

Figura 2.3: Uma árvore de nós

2.3.2 Cena

Uma cena é composta por um grupo de nós organizados de maneira hierárquica. Todas as cenas:

- Sempre têm apenas um nó raiz
- Podem ser salvas no disco e recarregadas posteriormente
- Podem ser instanciadas

Executar um jogo significa executar uma cena. Um projeto pode ter diversas cenas, portanto deve haver uma cena específica que deve ser executada inicialmente quando o jogo é aberto. O editor da *Godot* é basicamente um editor de cenas, com diversas interfaces para modificá-la e editar os nós que a compõe. Uma boa maneira de gerenciar os objetos que pertencem à um jogo é organizá-los em cenas.

2.3.3 Instanciação

É possível gerar cópias de uma cena A criando novas instâncias dela e adicionando a raiz de A como filha de um nó B de outra cena. Dessa maneira, uma cena pode ter várias outras cenas instanciadas nela. Um exemplo de como a instanciação de cenas é usada na prática: para criar uma sala com vários objetos nela, o *level designer* instancia os objetos na cena da sala pelo próprio editor da *Godot*. Se for um jogo em que o jogador tem uma arma de fogo, cada projétil disparado pela arma será instanciado na cena do ambiente. Porém, como essa instanciação é feita de maneira dinâmica durante o jogo, ou seja, depende do *input* do jogador, a cena dos projéteis deve ser previamente carregada e, no momento em que o jogador pressionar a tecla para disparar a arma, uma nova instância do projétil deve ser feita e adicionada na árvore.

2.3.4 Sinal e grupo

Sinais são notificações emitidas à partir de um nó, quando algum tipo determinado de evento ocorre, ou por meio de uma invocação de método. Um sinal pode estar conectado em um ou mais nós, que receberão sua emissão e executarão o método que está associado. É possível também anexar argumentos ao sinal, que serão passados para o método a ser invocado. Existem sinais predefinidos para cada classe, mas há a possibilidade de criar novos sinais por código. Alguns eventos importantes só podem ser detectados por meio de sinais.

Nós podem ser adicionados em grupos, pela interface da *Godot* ou por código, e não existem restrições em relação à quantidade de grupos que um nó pertence. Grupos são úteis para verificar se um nó corresponde à um conjunto específico antes de executar algum bloco de código. Por exemplo, existe um sinal que é emitido quando um nó do tipo *Area2D*

colide com um nó do tipo *StaticBody2D, KinematicBody2D,* ou *RigidBody2D,* e o argumento do método é o nó do tipo *Area2D.* Como vários objetos do jogo podem ter um nó do tipo *Area2D,* é verificado por código se o nó em questão está contido em algum grupo, por exemplo o grupo de inimigos do jogo. Uma funcionalidade exclusiva dos grupos é ter a possibilidade de chamar um método que está contido em todos os nós presentes em algum grupo, ou até mesmo editar as propriedades desses nós, sem precisar iterar pela árvore inteira em busca dos nós que estão contidos nesse grupo e invocar individualmente o métodos para cada nó.

2.3.5 Herança

Na *Godot, scripts,* que são arquivados no disco, são considerados classes. Novas classes podem herdar de uma classe global, uma classe arquivada em disco (*script*), ou uma classe definida em um *script,* tornando-se uma extensão da classe base. Herança múltipla não é permitida, e métodos definidos nas classes estendidas substituem os métodos na classe base que têm o mesmo nome, mas ainda assim é possível invocar o método especificamente da classe base.

Existe ainda o conceito de cena herdada. É interessante criar cenas que sejam uma base para outras cenas que podem ser classificadas de uma maneira semelhante. Por exemplo, criar uma cena base para inimigos, e a partir dessa criar cenas herdadas dessa cena base, uma para cada inimigo. Dessa maneira, não é necessário replicar todas as características presentes em todos os inimigos em todas as cenas, já que elas se concentram apenas na cena base, e a modificação delas será refletida em todas as cenas herdadas, criando uma alusão análoga à orientação a objetos.

2.3.6 *SceneTree* e *Singleton*

A única instância que é executada no início é a classe *OS.* Após isso, todos os *drivers*, servidores, linguagens, sistema de cenas, etc são carregados. Quando a inicialização é finalizada, é fornecida à classe *OS* uma classe *MainLoop,* que é onde o jogo é iniciado. Se essa etapa não for realizada, o processo é finalizado. A classe *MainLoop* contém alguns métodos para inicialização, *callbacks* e *input.* No momento que essa classe é providenciada para a classe *OS,* uma classe *SceneTree* é criada. É importante notar que essa classe existe por alguns motivos:

- Contém a raiz da classe *Viewport*, que é onde uma cena é adicionada como filha quando é aberta pela primeira vez para se tornar parte da *SceneTree*
- Contém informação sobre os grupos, e tem os meios para invocar todos os nós em um grupo ou fornecer uma lista deles

- Contém funcionalidades do estado atual do jogo, como a configuração de pausa e encerramento de processos

A classe *SceneTree* gerencia as cenas e a hierarquia dos nós em cada uma delas. Nós podem ser adicionados, recuperados e removidos, e a árvore da cena pode ser pausada. Cenas podem ser carregadas, trocadas e recarregadas. Quando um nó entra na *SceneTree,* ele se torna ativo, e ganha acesso à tudo relacionado aos processos, *input,* exibição de gráficos 2D e 3D, notificações, grupos, etc. Quando são removidos da *SceneTree,* perdem o acesso. Os passos a seguir definem esse procedimento:

1. Uma cena é carregada do disco ou criada por meio de código
2. A raiz dessa cena é adicionada como filha de *Viewport,* ou como filha de qualquer filha de *Viewport*
3. Todos os nós dessa cena irão receber uma notificação "*entre_ tree*", na ordem de cima para baixo (primeiro os pais, depois os filhos)
4. Quando um nó e todos os seus filhos estão dentro da cena ativa, esse nó recebe uma notificação "*ready*"
5. Quando uma cena (ou parte dela) é removida, uma notificação "*exit_ tree*" é enviada é enviada na ordem de baixo para cima (primeiro os filhos, depois os pais)

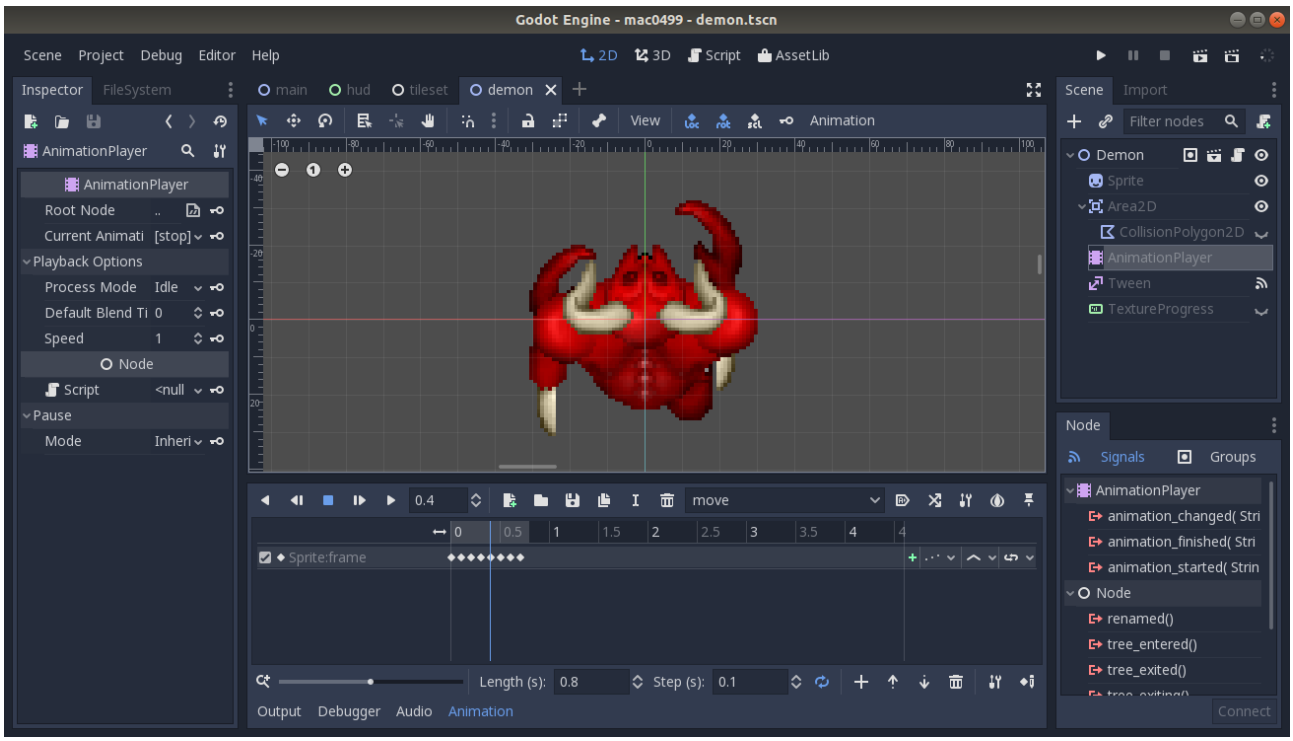

Figura 2.4: Interface da *Godot*

Na esquerda da Figura 2.4, a aba *Inspector,* onde é possível editar os atributos da classe *AnimationPlayer.* Na direita, a aba *Scene,* que exibe a árvore de nós, e a aba *Node,* que está exibindo alguns sinais nativos. No rodapé, ao centro, a aba *Animation,* utilizada para criar animações.

Em algumas situações, é necessário guardar informações persistentes que serão utilizadas após uma mudança de cena, por exemplo dados que salvam o progresso do jogador, ou o inventário de um personagem. Para isso, é possível criar nós que atuam como *singletons* e associar *scripts* à eles. Esses nós serão carregados na inicialização do jogo, independente da cena que será executada, e sempre são adicionados como filhos do *Viewport* antes que outras cenas sejam carregadas. As variáveis presentes nos *scripts* são como variáveis globais, que podem ser acessadas por todas as cenas a qualquer instante. Além dessas características, o nó *singleton* será responsável por lidar com transições entre cenas, como no caso onde existe uma tela de carregamento do jogo.

2.3.7 *GDScript*

GDScript é uma linguagem de programação dinâmica de alto nível. Tem uma sintaxe semelhante à *Python,* isto é, blocos de código são baseados na indentação, e muitas palavras chave são parecidas. Seu objetivo é ser otimizado e altamente integrado com a *Godot Engine,* permitindo flexibilidade para a criação e integração de conteúdo. Além dos tipos de dados básicos que estão embutidos, existem tipos de dados relacionados à vetores e à própria *engine* [Linietsky, 2018]. Eles são:

- *Vector2*: vetor 2D que contém os campos x e y. Pode ser acessado como uma variável indexada.
- *Rect2*: retângulo 2D que contém dois vetores como campos: *position* e *size.*
- *Vector3*: vetor 3D que contém os campos x, y e z. Também pode ser acessado como uma variável indexada.
- *Transform2D*: matriz 3x2 usada em transformações de duas dimensões.
- *Plane*: plano 3D em formato normalizado. Contém um vetor *normal* e um distância escalar *d.*
- *Quat*: um quaternião é um tipo de dado utilizado para representar rotações em 3 dimensões. É útil para interpolar rotações.
- *AABB*: caixa delimitadora alinhada aos eixos. Contém os vetores *position* e *size.*
- *Basis*: matriz 3x3 usada em rotações 3D e dimensionamento. Contém 3 vetores como campos (x, y e z) e também pode ser acessada como uma variável indexada de vetores 3D.
- *Transform*: representa uma transformação em 3 dimensões. Contém um *Basis,* chamado *basis,* e um *Vector3,* chamado *origin.*

- *Color*: tipo de dado que representa uma cor. Contém 4 campos *float*: *r, g, b* e *a* (vermelho, verde, azul e o componente *alpha*). Também pode ser acessado como *h, s* e *v* (matiz, saturação e valor).
- *NodePath*: caminho compilado para um nó, usado principalmente no sistema de cenas. Pode ser facilmente atribuído de e para uma String.
- *RID*: identificação de um recurso. Servidores usam *RID*s genéricos para referenciar dados opacos.
- *Object*: classe base para tudo que não é um tipo de dado embutido.

Variáveis podem ser exportadas, isto é, elas podem ter o seu valor salvo junto com a cena em que estão presentes. Elas também estarão disponíveis para serem modificadas no editor de propriedades, que é um dos principais motivos para se exportar uma variável. A razão para isso é a facilidade de integração do trabalho com desenvolvedores que não são programadores e não se sentem confortáveis escrevendo código, pois eles podem balancear os valores dos atributos pelo próprio editor.

Se uma classe é herdada da classe *Reference,* então a memória ocupada por suas instâncias serão liberadas quando não estiverem mais em uso. Não existem coletores de lixo, apenas contadores de referências. Por padrão, todas as classes que não definem a herança estendem da classe *Reference.* Caso contrário, a classe deve herdar da classe *Object* manualmente, e deve invocar o método *free.* Para evitar referências cíclicas cuja memória não pode ser liberada, existe um método chamado *weakref,* que fornece referências fracas.

GDScript oferece suporte à corotinas, através do método embutido *yield.* A invocação desse método irá retornar da função imediatamente, e o valor devolvido será o estado atual dessa função. Ao invocar o método *resume* nesse objeto, a função será executada a partir da última instrução. Uma integração que torna o uso do *yield* muito mais interessante é com sinais, pois esse método aceita dois parâmetros: um objeto e um sinal. Quando o sinal em específico for emitido desse objeto, a execução será retomada.

Capítulo 3

Proposta

Path of The Wicked é um jogo eletrônico do gênero *Tower Defense,* desenvolvido inteiramente pelo autor desta monografia, utilizando a *Godot Engine*. Sua concepção foi inspirada no jogo *GemCraft – Chasing Shadows*. *Path of the Wicked* é organizado em ondas, onde um conjunto aleatório de *creeps* surge em diferentes pontos fixos do mapa, em intervalos de tempo iguais. A cada onda que o jogador sobrevive, novos *creeps* mais difíceis vão surgindo, e a quantidade aumenta progressivamente. O início de uma nova onda não depende se os *creeps* da onda anterior foram abatidos, ou seja, assim que uma onda termina, é iniciado uma contagem regressiva para o início da próxima onda. Existem 25 tipos diferentes de *creeps*, cada um com os seguintes atributos: pontos de vida, velocidade, fraqueza e resistência às cores de cristais. Os *creeps* se movem em direção à base, seguindo um algoritmo de busca. Se os pontos de vida de um *creep* chegam à zero, esse *creep* morre e fornece ao jogador uma quantidade de dinheiro, proporcional à dificuldade do *creep*, que representa sua única fonte de recursos, utilizado para fazer compras.

Se um *creep* atinge a base, ele morre e o jogador perde dinheiro, e se o dinheiro do jogador acabar, ele perde a partida. Para que o jogador se proteja, ele deve comprar torres e cristais. As torres só podem ser colocadas em células do mapa que não possuem nenhum caminho, e têm um alcance fixo, determinado por um círculo. Torres podem ou não estar equipadas com cristais. Caso estejam, os cristais iram atirar projéteis nos *creeps*, que retiram uma determinada quantidade de pontos de vida, dependendo do tipo do cristal. Quando um cristal é equipado em uma torre, ela não pode atirar durante um pequeno período. O jogo possui 6 tipos de cristais, cada um com 6 cores diferentes. Se um *creep* tem fraqueza à uma cor de cristal, ele sofre o dobro de dano de projéteis originados de cristais dessa cor, e se tem resistência, o dano diminui pela metade. Cada projétil atirado de um cristal tem uma chance de causar um efeito ao *creep* ou gerar algum tipo de vantagem para o jogador, dependendo da cor do cristal. A relação dos efeitos e das cores é a seguinte:

- Amarelo: Bônus de recursos. Pode fornecer uma quantia em dinheiro ao jogador, mesmo antes do *creep* ser abatido.
- Rosa: Dano crítico. Pode causar o dobro de dano em um *creep.*

- Verde: Envenenamento. O *creep* envenenado sofre danos intervalados durante um período.
- Transparente: Paralisia. Um *creep* paralisado não pode se mover enquanto este efeito durar.
- Azul: Lentidão. O *creep* se move com velocidade reduzida durante alguns segundos.
- Vermelho: Dano em área. *Creeps* que estão próximos do local de colisão sofrem uma quantia de dano.

Um projétil pode ter seu alvo substituído, caso o *creep* que esteja seguindo seja eliminado. Esse critério depende da existência de outro *creep* dentro da área de alcance da torre cujo projétil teve origem. A câmera do jogo tem dois níveis de *zoom,* um que mostra com detalhes alguma parte do mapa, e dá liberdade ao jogador navegar por ele, e outro que exibe uma visão do mapa inteiro. A HUD (*head-up display*) do jogo mostra em qual onda o jogador está ou o tempo até que a próxima onda se inicie, a quantidade de dinheiro, o inventário ou o menu de compras. O inventário é o local onde o jogador guarda os cristais que foram comprados e que não estão equipados em nenhuma torre. Existe um botão que mostra o menu de compras, onde o jogador pode comprar os cristais e a torre, e uma área de descarte, em que o jogador pode descartar cristais caso não vá mais utilizá-los ou se o inventário estiver cheio. O jogo possui atalhos no teclado para comprar cristais e torres. A jogabilidade se resume em clicar em um cristal do inventário e arrastar para uma torre, equipando-a.

O mapa é composto de um *grid* hexagonal, e os caminhos têm início em pontos fixos, onde surgem os *creeps*, e terminam na base. Inicialmente é sorteado um número que representa o viés do caminho, ou seja, para qual direção irá tender. Para determinar a posição da próxima célula gerada do mapa, é calculado um vetor com origem no centro da célula que aponta para a base. Em seguida, esse vetor é rotacionado em um ângulo gerado por uma distribuição gaussiana, cujo desvio padrão é proporcional à distância da célula até a base, e a média é o viés do caminho.

Após a geração do mapa, é criado um grafo, em que cada célula é um nó e cada caminho que conecta duas células é uma aresta. Para cada ponto de surgimento dos *creeps* é calculado o caminho até a base usando o A*, um algoritmo de busca informada. Conforme o jogador equipa torres com cristais, os pesos dos nós do grafo são atualizados, e novos caminhos são calculados. As células desses caminhos são armazenadas em uma lista, e durante o jogo, cada *creep* acessa essas informações para saber qual será a próxima célula que irá se dirigir.

Capítulo 4

Desenvolvimento

Neste capítulo, será apresentado como foi o desenvolvimento e os detalhes de implementação do jogo *Path of The Wicked.*

4.1 Câmera

Existem dois níveis de *zoom* da câmera, que podem ser alternados usando a roda do mouse: o primeiro deles enquadra o mapa inteiro, e o segundo cobre 25% do mapa. No primeiro modo, não é possível navegar com a câmera, afinal a visão do mapa já é completa. No segundo modo, a navegação é feita pressionando o botão do meio do *mouse,* e movendo na direção contrária da qual se deseja observar. Os atributos da classe *Camera2D* que foram modificados para produzir esse efeitos são o *offset* e o *zoom,* ambos *Vector2.*

Vários cuidados foram tomados para evitar que o jogador pudesse observar alguma parte da cena que não contivesse alguma textura do *Tilemap.* No modo de zoom que a câmera está "longe", a janela não é redimensionável, e no que a câmera está "perto", a janela não pode ser dimensionada além de um limite, definido horizontalmente em 1920 *pixels* e verticalmente em 1080 *pixels.* Só é permitido afastar a câmera se o tamanho atual da janela for horizontalmente menor ou igual a 960 *pixels* e verticalmente menor ou igual a 540 *pixels.* O código a seguir é do método que atualiza o *offset* da câmera, invocado durante todos os *frames* que o jogador estiver pressionando o botão do meio do *mouse:*

```
1  func change_camera_offset():
2      var cur_pos = get_viewport().get_mouse_position()
3      var cur_offset = init_pos - cur_pos
4      if not free_camera:
5          var diff = MAX_WINDOW_SIZE - OS.window_size
6          if cur_offset.x < 0 or cur_offset.x > diff.x:
7              init_pos = Vector2(cur_pos.x + self.offset.x, init_pos.y)
8          if cur_offset.y < 0 or cur_offset.y > diff.y:
9              init_pos = Vector2(init_pos.x, cur_pos.y + self.offset.y)
10         cur_offset.x = min(max(0, cur_offset.x), diff.x)
11         cur_offset.y = min(max(0, cur_offset.y), diff.y)
12     self.offset = cur_offset
```

4.2 HUD

De imediato, já é importante notar que a raiz da HUD (*head-up display*) é filha da câmera, ou seja, ela herda sua posição e portanto a acompanha sempre que o jogador a move. No canto superior esquerdo, está presente um nó do tipo *Label,* que é uma caixa de texto, e um nó do tipo *TextureProgress,* usado neste caso como uma simples barra de progresso, indicando o tempo que falta até o início da próxima onda. Essa barra possui duas texturas, uma escura e uma clara, que interpolam seus valores e exibem percentualmente o tempo decorrido.

Figura 4.1: O inventário do jogador com alguns cristais e um *popup* exibindo informações, e o menu de compras

Na margem direita da tela, é exibido um painel que contém o inventário do jogador, um botão de compras e uma área de descarte de cristais. Quando o jogador aperta o botão de compras, o inventário some e dá lugar à alguns botões que representam os diferentes tipos de cristais e a torre. O jogador só pode escolher qual tipo quer comprar, ou seja, a cor do cristal é aleatória, com probabilidade uniforme, e não é possível comprar novos cristais caso o inventário já esteja cheio. Nesse caso, para liberar um espaço no inventário, o jogador pode escolher entre equipar uma torre com um cristal ou descartar um cristal selecionando e arrastando para a área de descarte. Existem atalhos que facilitam a compra de cristais e da torre: as teclas de “1” à “6” compram um cristal do tipo correspondente, e a tecla “Espaço” compra uma torre.

No canto superior direito, existe um contador de dinheiro do jogador. Como a dinâmica do jogo se torna mais rápida a cada onda, a legibilidade do contador pode ficar comprometida, pois são muitos *creeps* sendo abatidos simultaneamente. Para evitar isso, foi implementada uma animação que mostra durante 1 segundo a quantidade acumulada que deve atualizar o contador, e somente após esse tempo o contador de fato é atualizado. Os métodos a seguir tratam esse processo de animação e atualização do dinheiro:

```
1  func update_gold(amount):
2      gold += amount
3      gathered += amount
4      if gold <= 0:
5          main.game_over()
6      if gathered_label == null:
7          gathered_label = gold_label.duplicate(DUPLICATE_USE_INSTANCING)
8          gathered_label.margin_top = 48
9          self.add_child_below_node(gold_label, gathered_label)
10         gold_timer.start()
11     gathered_label.set_text('%+d' % gathered)
12
13 func _on_GoldTimer_timeout():
14     var gold_tween = gold_label.get_node('GoldTween')
15     tween_label = gathered_label.duplicate(DUPLICATE_USE_INSTANCING)
16     self.add_child_below_node(gold_label, tween_label)
17     gold_tween.interpolate_property(tween_label, 'margin_top', \
18             48, 16, .5, Tween.TRANS_LINEAR, Tween.EASE_IN)
19     gold_tween.interpolate_property(tween_label, 'self_modulate', \
20             Color(1, 1, 1, 1), Color(1, 1, 1, 0), .5, \
21             Tween.TRANS_LINEAR, Tween.EASE_IN)
22     gold_tween.start()
23     gold_timer.stop()
24     gathered_label.queue_free()
25     gathered_label = null
26     gathered = 0
27
28 func _on_GoldTween_tween_completed(object, key):
29     if key == ':margin_top':
30         tween_label.queue_free()
31         gold_label.set_text('Gold: %d' % (gold - gathered))
```

Também como parte da HUD, existe um *popup* informativo (ilustrado na Figura 4.1) que aparece sempre que o jogador passa o ponteiro do *mouse* em cima de um cristal ou de uma torre. Quando o cristal ou a torre estão no menu de compras, o *popup* informa o tipo, dano e preço, no caso dos cristais, e alcance e preço, no caso da torre. Quando um cristal está no inventário, o *popup* informa a cor, tipo, efeito e dano, e quando uma torre está equipada com um cristal, o *popup* informa o tipo, efeito e dano do cristal, e o alcance da torre. A implementação do *popup* teve detalhes bem sutis, começando pela posição que ele surge.

Naturalmente, o lugar mais comum é do lado esquerdo, com sua extensão para baixo, mas quando o objeto está próximo dos limites da janela, o *popup* teria uma parte que ficaria cortada. Para evitar esse inconveniente, bastou deslocar o *popup* sempre que fosse identificada essa situação. Porém, um problema mais grave que esse surgiu de imediato: com o deslocamento, era possível que o *popup* ficasse por cima do cristal, e isso impediria qualquer interação do jogador com o objeto, pois o *popup* herda da classe *Panel,* que por sua vez herda da classe *Control,* cujos atributos não permitem a identificação de *input* por sinais em um *canvas* que esteja por trás dele.

A correção desse problema foi simples: sempre que o ponteiro do *mouse* passar por um *popup,* este some, afinal a jogabilidade é muito mais importante do que o *popup* automático nesse caso. Por fim, no centro da tela, existe um nó do tipo *Label,* utilizado para exibir notificações ao jogador.

4.3 Cursor

O cursor é uma entidade do jogo sem representação visual, responsável pela transferência de cristais. Essa transferência pode ser feita entre torres e espaços do inventário, e é caracterizada como uma movimentação ou uma troca, caso o alvo esteja ocupado por um cristal. É o cursor que garante se o alvo é válido ou não, que guarda a informação de onde foi retirado o cristal, e promove a troca de posição de cristais caso seja necessário. O cursor, as torres, os espaços do inventário, e a caixa de descarte possuem um nó do tipo *Area2D,* que emite um sinal quando outros nós desse tipo entram ou saem dele, e é dessa forma que o mapeamento do alvo é feito. Se o jogador soltar o botão esquerdo do *mouse* sem um alvo definido, o cristal retorna para a posição de origem.

O cursor tem sempre uma instância única, que é criada sempre que o jogador pressiona o botão esquerdo do *mouse* e é removida da árvore quando o jogador solta esse botão, realizando a ação apropriada com o cristal atualmente selecionado. Existe um atalho que remove automaticamente o cristal de uma torre e a transfere para o inventário: basta que o jogador pressione o botão direito do *mouse* em alguma torre equipada com um cristal. Quando um cristal é selecionado, ele muda de posição na árvore: normalmente ele é filho da torre ou de um espaço no inventário, porém quando a instância do cursor é criada, o cristal se torna filho do cursor, e portanto herda sua posição, que é atualizada todos os *frames* com a posição real do ponteiro do *mouse.* Os métodos a seguir são responsáveis por promover a transferência de cristais.

```
func _input(event):
    if event.is_class('InputEventMouseMotion'):
        self.position = get_global_mouse_position()
    elif event.is_action_released('ui_select'):
        self.remove_child(gem)
        gem.scale = Vector2(1, 1)
        if target == null:
            move_gem()
        else:
            swap_gems()
        if source.is_in_group('slot') and source != target:
            source._on_Slot_mouse_exited()
        get_parent().cursor = null
        self.queue_free()

func swap_gems():
    var gem_on_target = target.gem
    if gem_on_target != null:
        target.remove_child(gem_on_target)
        move_gem(source, gem_on_target)
    move_gem(target)

func move_gem(node = source, _gem = gem):
    node.add_child(_gem)
    node.gem = _gem
    if node.is_in_group('slot'):
        _gem.position = node.offset
    elif node.is_in_group('tower'):
        node.start_cooldown()
        _gem.position = Vector2(0, 0)
    elif node.is_in_group('discard'):
        _gem.queue_free()
        node.gem = null
```

4.4 Torres

Durante a geração do grafo que mapeia o terreno, é armazenado também as coordenadas de todas as células do *Tilemap* que representam a grama, ou seja, os *tiles* que não possuem nenhum caminho. Essas coordenadas são utilizadas para criar as chamadas torres *dummy,* que são torres não funcionais, usadas apenas para indicar ao jogador quais são os locais em que é possível posicionar uma torre no momento da compra. Após essa ação, a torre *dummy* é removida da árvore (ela é neta do nó que representa o mapa), e na sua posição é criada uma torre funcional.

Para melhor identificar qual é o alcance efetivo das torres, quando o jogador passa o ponteiro do *mouse* por cima de alguma delas é desenhado um círculo centrado na torre, que representa a borda da área de cobertura. Enquanto essa área contiver algum *creep,* o cristal equipado irá atirar projéteis. Sempre que o jogador equipar um cristal em uma torre, haverá um *cooldown* com duração de 1 segundo, em que o cristal não pode atirar

nem pode ser removido. Esse *cooldown* é apenas uma pequena penalidade, para que o jogador realize a ação apenas quando realmente for necessário e não possa explorar de maneira abusiva.

Figura 4.2: Círculo que exibe a área de alcance de uma torre

Uma variável bastante importante que está contida no *script* da torre é a *nearby_ creeps.* Essa variável é um vetor que armazena todos os *creeps* que estão em contato com a área de alcance da torre, e é utilizado para definir qual será o alvo do cristal quando existem múltiplos inimigos, inclusive quando um projétil muda de alvo durante sua trajetória. Os métodos a seguir lidam com a inserção e a remoção de *creeps* desse vetor:

```
1  func _on_NearbyArea_area_entered(area):
2      var creep = area.get_parent()
3      if creep != null and creep.is_in_group('creep') and not creep.dying:
4          nearby_creeps.append(creep)
5          creep.towers.append(self)
6          if nearby_creeps.size() == 1 and not cooldown.visible:
7              if gem != null and gem.timer.time_left == 0:
8                  gem.shoot()
9
10 func _on_NearbyArea_area_exited(area):
11     var creep = area.get_parent()
12     if creep != null and creep.is_in_group('creep'):
13         nearby_creeps.erase(creep)
```

4.5 *Creeps*

	HP	VEL	WEAKNESS	STRENGTH
RAT	44	50	RED	CLEAR
BANDIT	97	30	BLUE	CLEAR
BEE	65	60	BLUE	GREEN
SKELETON_MAGE	132	40	RED	YELLOW
TIMBERMAN	293	25	YELLOW	BLUE
SKELETON_KNIGHT	256	40	RED	PINK
HORNET	275	45	GREEN	BLUE
KLIVER	358	45	GREEN	CLEAR
SCORPION	391	50	RED	GREEN
OGRE	1220	20	PINK	YELLOW
COCKROACH	738	40	BLUE	RED
BLUE_MAGE	1031	35	PINK	GREEN
GREEN_MAGE	1191	35	PINK	RED
CRAWLER	958	50	CLEAR	YELLOW
OGRILLION	2783	20	PINK	BLUE
TROLL	3288	20	YELLOW	CLEAR
CHITINIAC	1745	45	BLUE	PINK
WOLFBEAST	2715	35	YELLOW	PINK
UGONOTH	2959	35	YELLOW	RED
GARGANT_SIMPLE	3511	35	GREEN	YELLOW
GARGANT_BERSERKER	4595	30	CLEAR	GREEN
GARGANT_LORD	5406	30	GREEN	RED
GARGANT_BOSS	5780	30	CLEAR	PINK
DRAGON	4010	50	CLEAR	BLUE
DEMON	7284	30	----------	----------

Tabela 4.1: Relação dos atributos de cada *creep* presente no jogo

As informações sobre cada creep, ou seja, os pontos de vida, velocidade, fraqueza e resistência são armazenados em um dicionário, cuja chave é uma string contendo o nome do creep e o valor é uma lista com essas informações. O valor de um *creep* é o número de pontos de vida vezes a velocidade dividido por 40, e indica numa escala numérica a dificuldade desse *creep*. A quantidade de dinheiro que cada *creep* fornece ao jogador quando é abatido é igual ao seu valor, e a quantidade de dinheiro que o jogador perde quando um *creep* atinge a base é o seu valor vezes 3 vezes a razão entre os pontos de vida restantes e os pontos de vida originais.

É durante a geração do grafo que mapeia o terreno que os pontos onde nascem os *creeps* são definidos. Esses pontos são as células que não possuem nenhum *tile* associado, ou seja, as células que não possuem textura, e que são adjancentes às células com um caminho nessa direção. Por motivos visuais, cada *creep* tem um deslocamento aleatório, dentro de um intervalo, na posição de origem, para que haja uma menor sobreposição entre eles e que ocupem melhor os caminhos.

Figura 4.3: Só é possível observar um *creep* imediatamente após seu surgimento no mapa no modo de depuração, em que a movimentação da câmera não é limitada pelas margens do terreno

Para que o jogo seja balanceado, os *creeps* mais difíceis não surgem nas primeiras ondas da partida, e são desbloqueados progressivamente. Para isso, existe um intervalo limitado em que cada *creep* que surge no mapa é aleatório com probabilidade uniforme, e esse intervalo cresce conforme o número respectivo que representa cada onda. O método utilizado para definir quando uma onda acaba, ou seja, quando param de surgir *creeps* e começa a contagem regressiva para a próxima onda, não é pela quantidade de inimigos, e sim como segue: cada onda tem uma quantidade de pontos correspondente. No início da onda, existe um contador que começa em zero, e a cada *creep* que nasce, o contador é incrementado com o valor desse *creep.* Quando o contador ultrapassa a quantidade de pontos associada à onda, é interrompido o processo e a onda acaba.

O cálculo da movimentação dos *creeps* é feito de maneira discreta, e animado de maneira interpolada. Isso significa que os *creeps* andam sempre do centro de uma célula até o centro da próxima célula, onde calculam para onde será o próximo destino. A interpolação é feita utilizando uma classe chamada *Tween* (o nome é uma referência à uma técnica de animação conhecida como *inbetweening*), mais especificamente o método *interpolate_property.* Esse método recebe, dentre outros argumentos, a posição atual, a posição final, e a duração da animação. Dessa forma, é feita a interpolação entre esses valores com relação ao tempo total e a duração de cada *frame.*

A morte de um *creep* é um evento que precisa ser tratado com cautela, pois existem algumas referências em vários *scripts* que devem ser alteradas. Como cada torre tem um vetor que contém os *creeps* dentro da área de alcance, quando um *creep* morre ele deve se remover desse vetor. Após essa ação, deve ser decidido o que será feito com cada projétil que está indo em direção ao *creep* que acabou de morrer. Se existe algum outro *creep* dentro da área de alcance da torre da qual o projétil foi disparado, então o novo alvo desse projétil se torna o próximo *creep* contido no vetor de referências. Caso contrário, é criado um nó *dummy* no exato local onde o *creep* morreu, sem identicação visual, com uma área

de colisão idêntica, e que só colide com o projétil em questão. Para que não haja colisão de outros projéteis nessa área, é criado um grupo, cujo nome é o mesmo do projétil (cada projétil tem um nome único), que é exclusivo desse nó *dummy*. Por fim, a referência do alvo desse projétil é atualizada para esse nó. O código a seguir exibe o método que é invocado quando um *creep* morre:

```
1  func die():
2      dying = true
3      tween.stop_all()
4      hud.update_gold(self.value)
5      for tower in towers:
6          tower.nearby_creeps.erase(self)
7      for proj in projectiles_node.get_children():
8          if proj in projectiles.values():
9              if proj.tower.nearby_creeps.size() > 0:
10                 proj.creep = proj.tower.nearby_creeps[0]
11                 proj.creep.projectiles[proj.name] = proj
12             else:
13                 create_dummy_creep(proj)
14     self.get_node('Area2D').monitoring = false
15     if anim.has_animation('death'):
16         anim.play('death')
17         anim.playback_speed = 1
18         yield(anim, 'animation_finished')
19     self.queue_free()
20
21 func create_dummy_creep(proj):
22     var node = Node2D.new()
23     var node_area = self.get_node('Area2D').duplicate( \
24             DUPLICATE_USE_INSTANCING)
25     node.position = proj.creep.position
26     node.add_child(node_area)
27     node.add_to_group(proj.name)
28     get_parent().add_child(node)
29     proj.creep = node
```

4.6 Projéteis e efeitos de cristais

Para que o projétil tenha uma movimentação suave e gere a sensação visual de que está "seguindo" o *creep*, existe um vetor normalizado que é rotacionado todo *frame*, ou seja, ele é uma aproximação do vetor que tem início no projétil e fim no *creep*. O momento mais marcante dessa movimentação é quando o alvo do projétil muda durante sua trajetória, e é possível observar o vetor sendo atualizado aos poucos, gerando uma curva agradável.

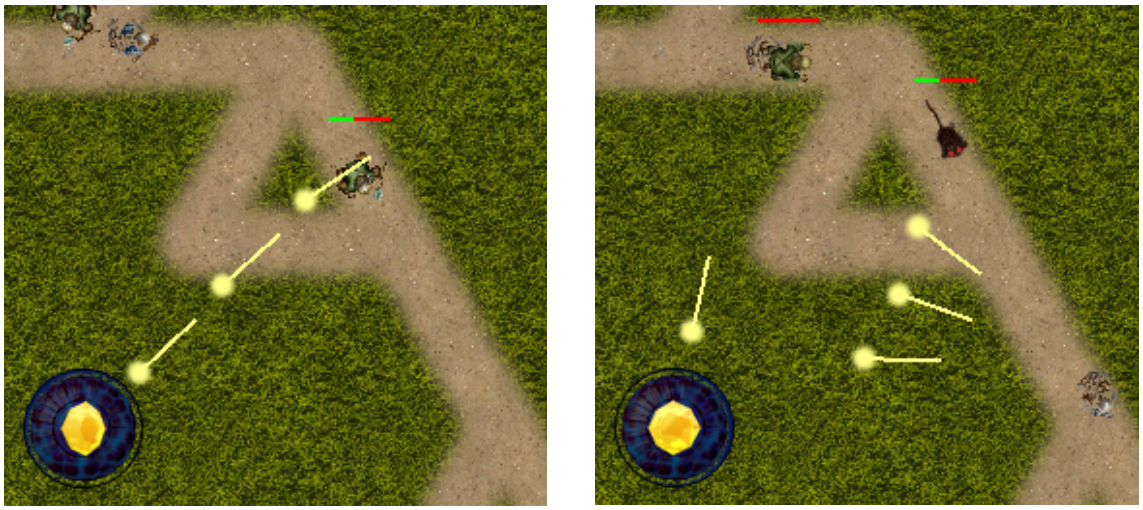

Figura 4.4: Comparação entre projéteis cujo alvo não foi alterado e projéteis que sofreram uma mudança repentina de alvo durante sua trajetória, com vetores indicando em qual direção o projétil se move, e visíveis apenas no modo de depuração

Como a atualização do vetor ocorre em todos os *frames,* os cálculos são feitos no método *_physics_process*:

```
1 func _ready():
2     self.z_index = 1
3     vector = creep.position - self.position * 2
4     vector = vector.normalized()
5
6 func _physics_process(delta):
7     angle = vector.angle_to(creep.position - self.position * 2)
8     vector = vector.rotated(angle * delta * 5)
9     self.position += vector * vel/10
```

Além do dano que os projéteis causam nos *creeps,* cada um possui um efeito dependendo da cor do cristal que o originou. Logo após a instanciação do projétil, é atribuído à ele qual *script* deve ser executado no momento que atingir um *creep*. O efeito *bonus gold,* correspondente ao cristal amarelo, fornece ao jogador uma quantia em dinheiro extra. Cada projétil que atinge um *creep* tem 20% de chance de o jogador ganhar 20% do valor do *creep* em dinheiro. O efeito *critical hit*, do cristal rosa, provoca um dano adicional aos pontos de vida do *creep.* Com 20% de chance, um projétil causa o dobro de dano no instante em que colide com um *creep.*

O efeito *poison,* do cristal verde, também provoca dano adicional, porém de maneira gradual. Com 1/3 de probabilidade, um projétil desse efeito pode causar envenenamento à um *creep.* Quando um *creep* está envenenado, ele não acumula esse efeito, ou seja, o efeito não pode ser intensificado caso ele continue sendo atingido por outros projéteis verdes. A duração do envenenamento é de 1 segundo, e a cada décimo de segundo, o *creep* sofre 10%

de dano correspondente ao projétil que o envenenou. O efeito *shock,* do cristal transparente, paraliza o *creep* atingido por 1 segundo, e o efeito *slow down,* do cristal azul, diminui a velocidade do *creep* pela metade durante 2 segundos. Ambos têm 1/3 de probabilidade de serem ativados, e cada um não pode ser acumulado.

O efeito *splash,* do cristal vermelho, cria uma área no formato de um círculo em volta do *creep* atingido, e todos os outros *creeps* que estiverem dentro dessa área recebem metade do dano que foi causado no *creep* atingido. Assim como a maioria dos efeitos, tem probabilidade de 1/3 a cada projétil que colide com um *creep.*

4.7 Bases de dados

Existem 3 bases de dados: a dos *creeps,* dos cristais, e dos *tiles* utilizados no mapa. A base de dados dos *creeps* contém um vetor com a cena de cada inimigo já carregada, pronta para ser instanciada, e o dicionário com as informações de cada *creep*. É utilizado no *script* que gerencia o surgimento dos *creeps* no mapa. A base de dados dos cristais contém um vetor de cenas carregadas para cada tipo de cristal, e dois dicionários: um com informações referentes à cor do cristal, e o outro com informações relacionadas ao tipo. Os vetores de cenas são utilizados no momento que o jogador compra um cristal, tornando simples o processo de adquirir um cristal aleatório de um tipo específico. O dicionário com informações sobre as cores contém o nome, a cor, o efeito e o *script* correspondente. O nome e o efeito são usados nos *popups* informativos, e a cor é utilizada para alterar o projétil no seu surgimento. O dicionário com informações sobre o tipo de cristal contém o dano recebido pelos *creeps* e o preço de cada tipo.

A base de dados dos *tiles* contém um *enum* que atribui uma constante para o ID de cada *tile,* de modo a tornar mais legível o código da geração do mapa, que manipula bastante as células do *Tilemap*. Contém também 6 vetores auxiliares, um para cada direção da célula hexagonal. Por exemplo, o vetor R (que indica um caminho para a direita), contém o número de identificação de todas as células que possuem um caminho para a direita, totalizando 15. Esses vetores são utilizados no mapeamento dos caminhos, que geram o grafo utilizado no algoritmo de busca.

4.8 *Shaders*

Foram desenvolvidos dois *shaders* para o jogo: um *shader* que borra a tela (*blur*), e outro que renderiza uma silhueta do objeto preenchida com uma cor sólida. O *shader* que borra a tela é utilizado quando a partida se encerra, ao mesmo tempo que o jogo é pausado. A disposição dos nós na árvore permite que as notificações sejam exibidas sem serem afetadas por esse *shader.*

Figura 4.5: *Blur shader,* exibido após o término de uma partida

O código a seguir está escrito na linguagem de *shading* da *Godot,* bem semelhante à GLSL ES 3.0. O valor de *amount* utilizado no jogo foi 2:

```
1 shader_type canvas_item;
2 uniform float amount : hint_range(0, 5);
3
4 void fragment() {
5     COLOR.rgb = textureLod(SCREEN_TEXTURE, SCREEN_UV, amount).rgb;
6 }
```

O *shader* de silhueta foi utilizado nos cristais que estão no menu de compras. O motivo do uso é que o jogador só pode escolher o tipo do cristal que irá comprar, pois a cor é aleatória. Dessa forma, esses cristais só exibem os seus formatos, não revelando qual será a cor adquirida no momento da compra.

Figura 4.6: Cristais no inventário, com cores, e cristais no menu de compras, com o *shader* de silhueta

O valor de *modulate* é a cor branca:

```
1 shader_type canvas_item;
2 render_mode blend_mix;
3 uniform vec4 modulate : hint_color;
4
5 void fragment() {
6     COLOR = vec4(modulate.rgb, texture(TEXTURE, UV).a * modulate.a);
7 }
```

Capítulo 5

Geração Procedural

Neste capítulo, será apresentado como foi feita a geração do mapa.

5.1 Células e *grid* hexagonal

As células que formam o mapa podem ter 2 ou 3 caminhos, com exceção da célula que representa a grama, que não possui nenhum caminho. Células com 2 caminhos são células normais, e células com 3 caminhos são células ramificadas. Cada célula possui um número de identificação, utilizado principalmente para atribuir uma célula à uma posição da matriz que representa o *Tilemap*. A figura a seguir exibe todas as 36 células existentes. O número de identificação de cada célula é obtido somando o número da linha com o da coluna.

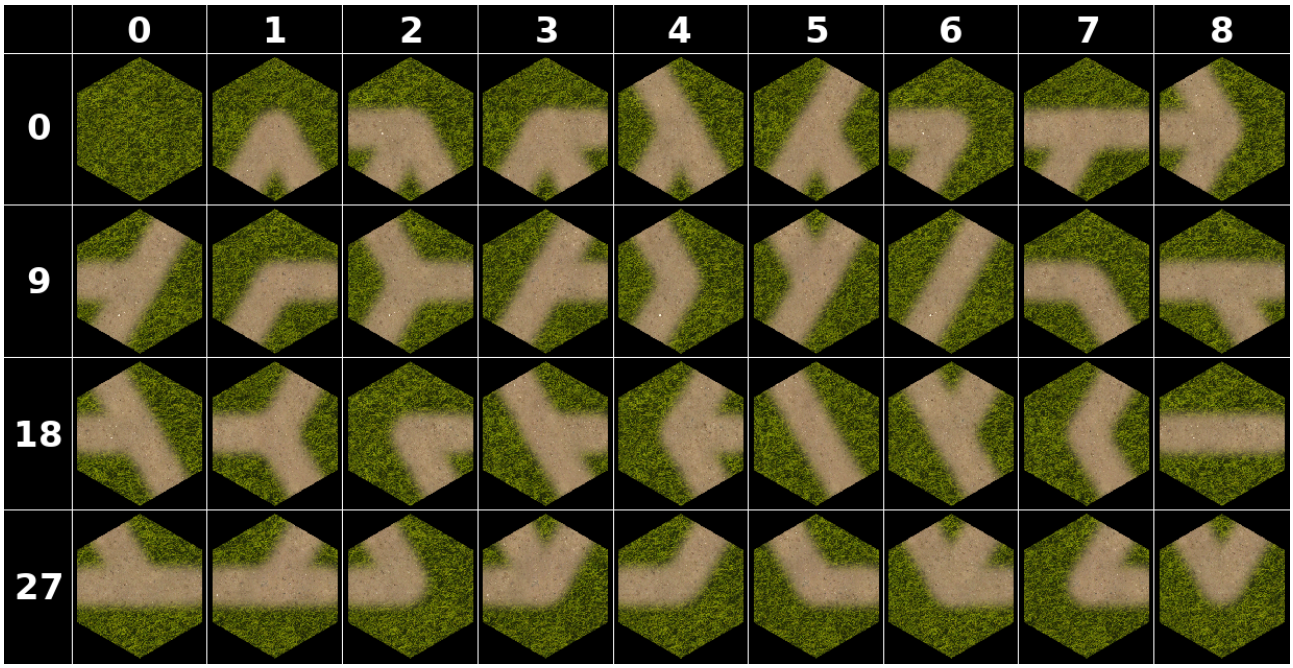

Figura 5.1: Células e seus números de identificação

A orientação escolhida para as células foi a linha horizontal, ou seja, o topo das células não é plano [Patel, 2018]. Como a matriz do *Tilemap* possui duas dimensões, o leiaute horizontal deve ter um deslocamento para a direita das linhas pares ou ímpares [Patel, 2018]. No caso em específico, foi decidido que as linhas ímpares seriam deslocadas. Logo, a aparência das células na matriz fica como segue:

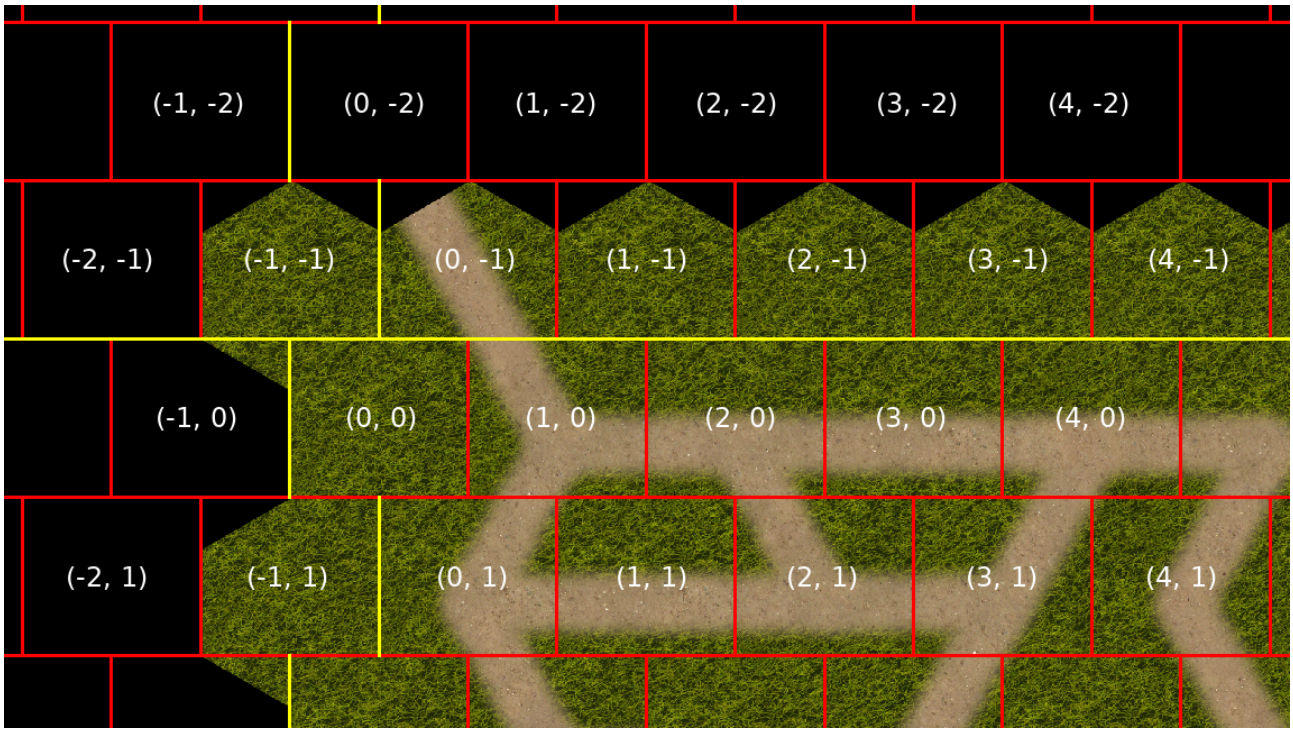

Figura 5.2: Disposição das células na matriz do *Tilemap*, com as posições correspondentes

É possível notar que a célula hexagonal possui a mesma área do quadrado em que está localizada, mas não o preenche completamente. Esse arranjo ocorre pois as lacunas são ocupadas caso haja uma célula que esteja presente em cima, e significa que o mapa exibido na tela é menor do que a área ocupada pelas células, ou seja, existe uma margem que garante que nunca seja mostrado para o jogador alguma porção do terreno que não esteja ocupada por uma célula.

5.2 Caminhos iniciais

O primeiro passo é preencher toda a área do terreno com células de grama. O tamanho do terreno tem 16 células de extensão na horizontal e 11 na vertical. Existem dois pontos fixos no mapa em que nascem *creeps*, e um ponto fixo onde a base está localizada. Os caminhos iniciais são os dois caminhos que ligam cada ponto de nascimento dos *creeps* à base. Posteriormente, são gerados caminhos adicionais que têm início em células presentes nos caminhos iniciais, chamadas células válidas. Todos os caminhos, tanto os

iniciais quanto os adicionais, são gerados um por vez, do início ao fim, sem paralelismo. Essa decisão foi tomada pois, caso os caminhos iniciais fossem gerados simultaneamente, existiria a possibilidade deles se cruzarem antes de atingirem a base. Dessa maneira, não haveria conectividade entre os pontos de nascimento e a base, tornando o jogo impossível. Outro problema de conectividade ocorre quando um caminho cruza com si próprio, gerando um caminho fechado. Como esse problema é raro de acontecer, é mais simples e pouco custoso descartar esse caminho desde o início e gerar outro.

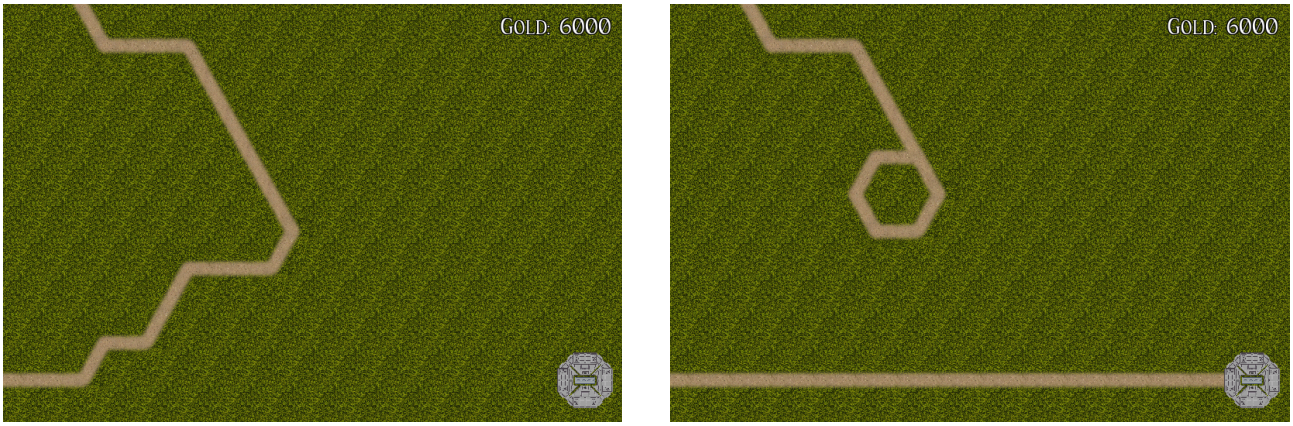

Figura 5.3: Problemas de conectividade nos caminhos iniciais

O primeiro caminho inicial é o que tem origem no topo. Para gerar um caminho inicial, é necessário ter a posição de origem na matriz do *Tilemap*, o número de identificação da primeira célula, o vetor geométrico que aponta de onde vem o caminho, e o viés do caminho. Para gerar uma nova célula, é calculado o vetor que tem início na célula atual e fim na base. Como o caminho a ser gerado é aleatório, é interessante que esse vetor seja rotacionado em um ângulo aleatório. Porém, seria ruim que esse valor fosse adquirido de uma distribuição uniforme, pois a informação sobre a direção da base seria perdida. Logo, esse valor é adquirido de uma distribuição gaussiana [Short, 2017], em que a média é o viés do caminho, e o desvio padrão é o comprimento normalizado do vetor. A garantia de conectividade se apresenta no fato de que, quanto mais próxima a célula está da base, menor é o desvio padrão, e portanto o valor aleatório é mais próximo da média [Millington, 2009].

Como a janela é um retângulo, que é um polígono convexo, qualquer vetor que tenha início e fim em pontos contidos nesse retângulo também estará contido no retângulo. Portanto, se o vetor rotacionado aleatoriamente for causar a geração de uma célula fora do mapa contido dentro da tela, o fator aleatório é descartado, e o resultado é o vetor calculado inicialmente. Para calcular o vetor geométrico de onde parte o caminho, primeiro é calculado o ângulo entre cada uma das seis direções do hexágono e o vetor resultante da rotação. O vetor geométrico é a direção cujo módulo do ângulo é o menor e não é igual à direção do vetor geométrico de entrada. Caso a célula a ser atribuída for em uma posição

da matriz que atualmente contém uma célula de grama, é obtido o número de identificação da célula a ser atribuída a partir dos dois vetores geométricos, o de entrada e o de saída.

Se essa posição da matriz contiver uma célula de 2 caminhos, é necessário ramificar essa célula, e para isso é utilizado apenas o vetor geométrico de entrada. A ramificação de um caminho determina também o seu fim. Porém, se nessa posição contiver uma célula ramificada, ou contiver uma célula que está presente no próprio caminho em processo de geração, configurando um caminho fechado, é efetuado o descarte das células desse caminho e as células antigas são restauradas. Por fim, se a posição da matriz coincide com a posição da base, então esse caminho está finalizado. A função a seguir gera uma nova célula:

```
func generate_tile(cell, bias, in_tile_dir):
    var vec = base_tile - tilemap.map_to_world(cell)
    var length = vec.length() / 3000
    if length < 0.1:
        length = 0
    var angle = vec.angle_to(Vector2(1, 0))
    var rand = gaussian(bias, length)
    var target_vector = Vector2(1, 0).rotated(angle).rotated(rand)
    var out_tile_dir = get_next_tile_direction(cell, \
            in_tile_dir, target_vector)
    if cell.y < 0 or cell.y >= 9 or cell.x < 0 or cell.x >= 13:
        out_tile_dir = get_next_tile_direction(cell, in_tile_dir, \
                Vector2(1, 0).rotated(angle))
    if tilemap.get_cellv(cell) == ts_db.GRASS:
        path_pos.append(cell)
        path_id.append(ts_db.GRASS)
        tilemap.set_cellv(cell, ts_db.get_tile_id(self, \
                in_tile_dir, out_tile_dir))
    else:
        if tilemap.get_cellv(cell) in ts_db.BRANCHED_TILE \
                or cell in path_pos:
            reset_path()
        else:
            path_pos.append(cell)
            path_id.append(tilemap.get_cellv(cell))
            invalid_cells.append(cell)
            tilemap.set_cellv(cell, branch(tilemap.get_cellv(cell), \
                    in_tile_dir))
            initial_path = false
        return [null, null]
    cell = get_cell(cell, out_tile_dir)
    if cell == base_pos:
        initial_path = false
        return [null, null]
    else:
        return [cell, get_in_tile_dir(out_tile_dir)]
```

5.3 Células válidas

Antes de gerar os caminhos adicionais, é necessário avaliar quais células são válidas e armazená-las em uma lista. As células válidas são utilizadas para iniciar um caminho adicional, e seguem 3 critérios: não é uma célula de grama, não é uma célula ramificada, e está dentro dos limites do terreno. Esse último serve para evitar que as células de entrada dos *creeps* no terreno sejam ramificadas. Se durante a geração dos caminhos adicionais a lista de células válidas ficar vazia, a geração do mapa é finalizada, independente da complexidade atingida. Algumas células válidas podem estar em uma posição que não seja possível iniciar nenhum caminho a partir delas. A figura a seguir mostra um exemplo em que esse caso ocorre:

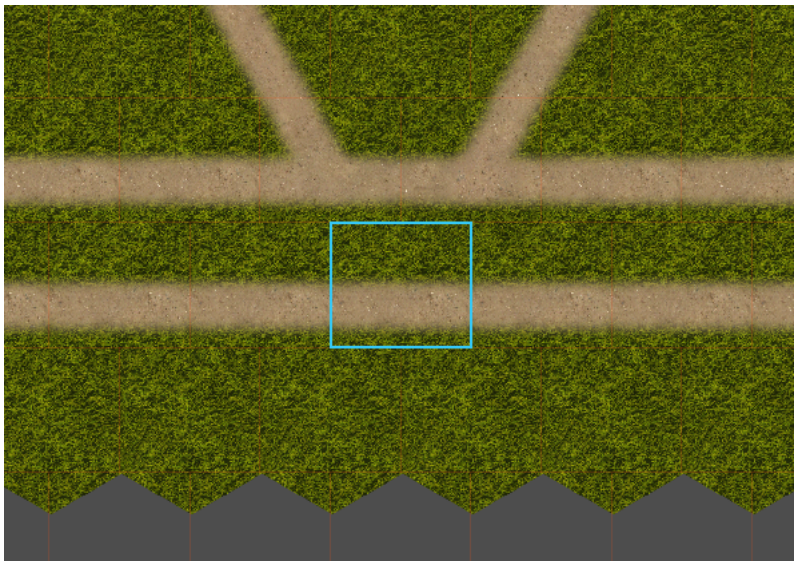

Figura 5.4: Uma célula válida que não pode iniciar um caminho

A célula evidenciada no exemplo não pode iniciar um caminho para a esquerda ou para a direita pois já existem conexões com outras células nessas direções. Também não pode iniciar um caminho para a esquerda inferior ou direita inferior, pois ultrapassaria os limites do terreno. Por fim, não pode iniciar para a esquerda superior nem direita superior, pois as células presentes nessas direções já são ramificadas. Esse problema só é identificado se esta célula for sorteada para iniciar um caminho adicional. A solução é interromper a geração desse caminho e remover esta célula da lista de células válidas. Durante a geração dos caminhos adicionais, a lista de células válidas é constantemente atualizada. Sempre que uma célula normal se torna ramificada, ele é removida da lista, e quando um caminho é validado, as células que passam nos 3 critérios mencionados anteriormente que estão presentes nesse caminho são adicionadas na lista.

5.4 Caminhos adicionais e complexidade do mapa

Depois que os dois caminhos iniciais foram gerados, e foi estabelecida a conexão entre os dois pontos de nascimento dos *creeps* e a base, uma série de caminhos adicionais são criados. Esses caminhos não são obrigatórios para tornar o jogo possível, mas foram implementados para tornar o mapa mais interessante e complexo, contribuindo também para evidenciar a inteligência artificial dos *creeps*. O número de caminhos adicionais por mapa não é fixo, pois cada caminho pode ter um comprimento diferente. Portanto, o critério utilizado para definir se um mapa está ou não finalizado é se a complexidade foi atingida. A complexidade do mapa é um valor numérico entre 0 e 1, e é a razão entre a quantidade de células normais e ramificadas e a quantidade de células no total.

O motivo desse cálculo é simples: as torres só podem ser posicionadas em células de grama, e quanto mais células com caminhos, maior é a área que deve ser coberta pelas torres. Logo, quanto menor a quantidade de células de grama, mais complexo é o mapa, e portanto a dificuldade aumenta [Yannakakis, 2018]. Após alguns testes, foi definido que o valor 0.4 para a complexidade é muito bom, pois proporciona um balanceamento bem justo para o jogo. Um caminho adicional começa de uma célula válida, que é sorteada aleatoriamente da lista. A partir dessa célula, é definida uma direção aleatória, entre as direções que são válidas. É importante ressaltar que durante a geração de todos os caminhos, a posição da matriz do *Tilemap* e o número de identificação de cada célula que é sobrescrita são armazenados. Se a avaliação do caminho constatar que o resultado não é válido ou não é interessante, o caminho é descartado e as células anteriores são restauradas.

O primeiro passo é ramificar a célula inicial utilizando a direção que foi obtida aleatoriamente. As células dos caminhos adicionais são geradas utilizando a mesma função que gera as células dos caminhos iniciais. O viés desse caminho adicional é um valor aleatório entre 0.2 e 2 ou -0.2 e -2. Não é muito interessante ter caminhos com viés muito próximo de zero, pois seria possível que a região ao redor da base ficasse muito congestionada, com poucas células de grama, e além disso, os caminhos iniciais já cumprem a função de direcionar os *creeps* para a base. Ademais, o viés pode assumir valores grandes, próximos de 2, caracterizando pouca restrição para a geração dos caminhos adicionais, que têm como principal função criar alternativas de rotas para os *creeps*. Se, após finalizado, um caminho adicional tiver comprimento menor do que 5 células, ele é descartado, pois é considerado que sua relevância para o mapa provavelmente é baixa [Short, 2017].

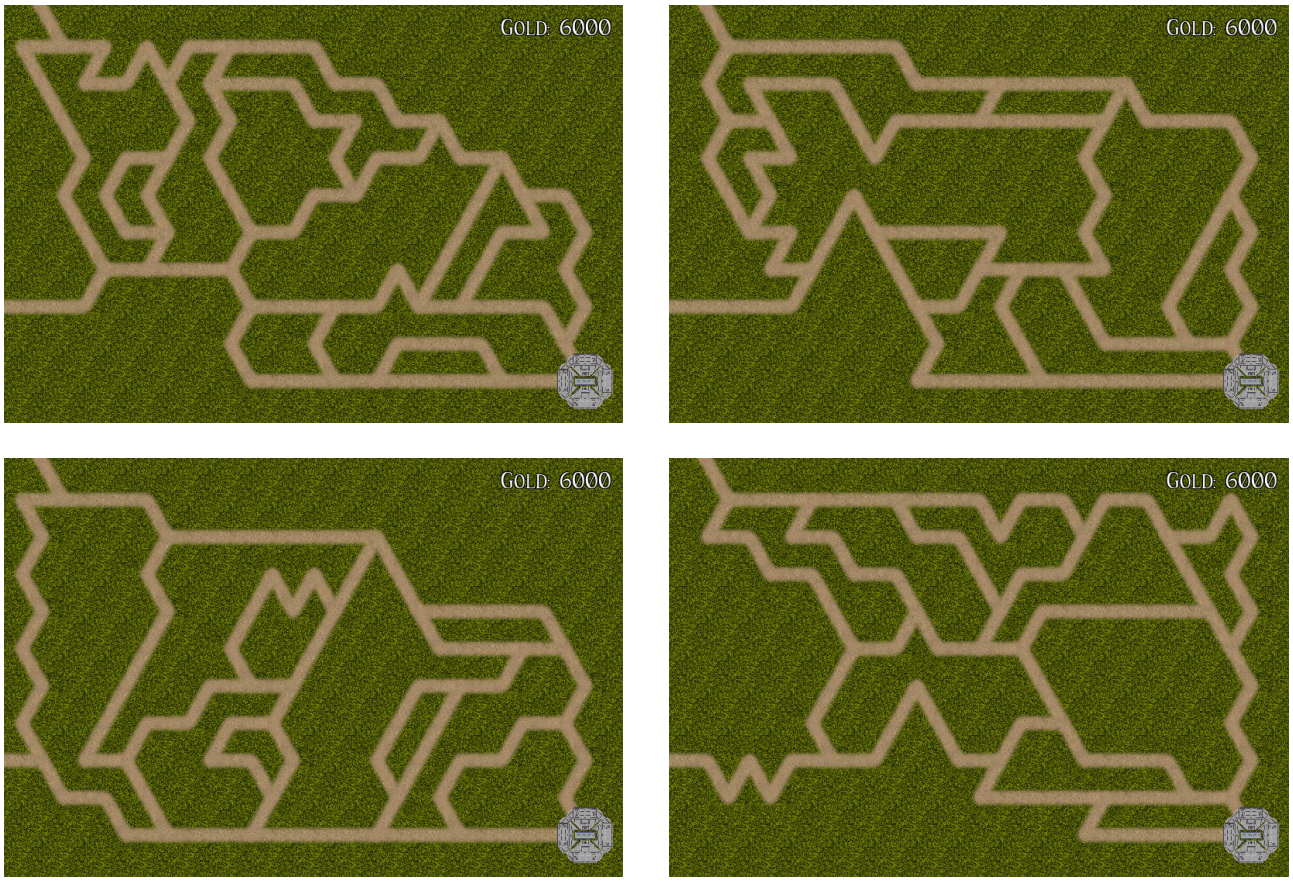

Figura 5.5: Exemplos de mapas gerados proceduralmente

Capítulo 6

Inteligência Artificial

Neste capítulo, será apresentado como foi implementada a inteligência artificial dos *creeps*.

6.1 Algoritmo A*

Com o mapa gerado, os *creeps* precisam calcular qual é a melhor rota até a base. Conhecido como *pathfinding*, é uma das áreas mais importantes em Inteligência Artificial aplicada em jogos. Apesar de existirem diversos algoritmos, a maior parte das aplicações são baseadas no algoritmo de Dijkstra e algumas de suas variações, como o algoritmo A* [Bourg, 2004][Millington, 2009]. Dado um grafo ponderado dirigido, o algoritmo de Dijkstra calcula o caminho de menor custo entre dois nós quaisquer desse grafo, utilizando uma fila de prioridades que armazena os nós visitados e expande eles em ordem crescente de custo atual acumulado. O algoritmo A* obtém o mesmo resultado, porém adiciona mais um fator nos cálculos chamado heurística. Se a heurística for boa, o A* se torna mais eficiente do que o algoritmo de Dijkstra [Millington, 2009], pois sua busca é muito mais direcionada para o objetivo, ou seja, não há desperdício de tempo investigando nós pouco promissores. No caso do jogo, a heurística é muito simples: a distância euclidiana até a base.

Para usar o algoritmo A*, é necessário mapear o terreno em um grafo ponderado dirigido [Millington, 2009]. Ao iterar pelas células, é criado um nó para cada célula normal ou ramificada [Millington, 2009], e armazenado a posição de cada célula de grama para posteriormente criar torres *dummy*. Para criar as arestas do grafo, é fundamental que seja identificado quais são as conexões de cada célula através dos caminhos. Cada aresta é criada utilizando um dicionário em que a chave é a posição de uma célula e o valor é uma lista de posições de células adjacentes. Esse dicionário utiliza os vetores que armazenam as células que possuem conexões pada cada um dos seis lados do hexágono do banco de dados do *Tileset* para verificar se a conexão é válida ou não. As conexões com a base e os pontos de nascimento dos *creeps* são tratadas de maneira especial: se uma célula possui uma conexão com uma célula de grama, então a posição dessa célula de grama é a posição da

base, e se uma célula possui uma conexão com uma célula vazia, então a posição dessa célula vazia é a posição de um ponto de nascimento de *creeps*.

Todas as arestas são direcionadas, ou seja, para cada dois nós A e B conectados do grafo, existe uma aresta direcionada de A para B e uma de B para A, a não ser que um desses nós seja a base ou um ponto de nascimento dos *creeps*. Nesses casos, se A for a base existe apenas uma aresta de B para A, e se A for um ponto de nascimento dos *creeps*, existe apenas uma aresta de A para B. A direção das arestas indica também em qual direção é permitida a movimentação de um *creep*. Na implementação utilizada, os pesos são colocados nos nós, e todos têm peso inicial 1. Na figura a seguir, é exibido o grafo no modo de depuração.

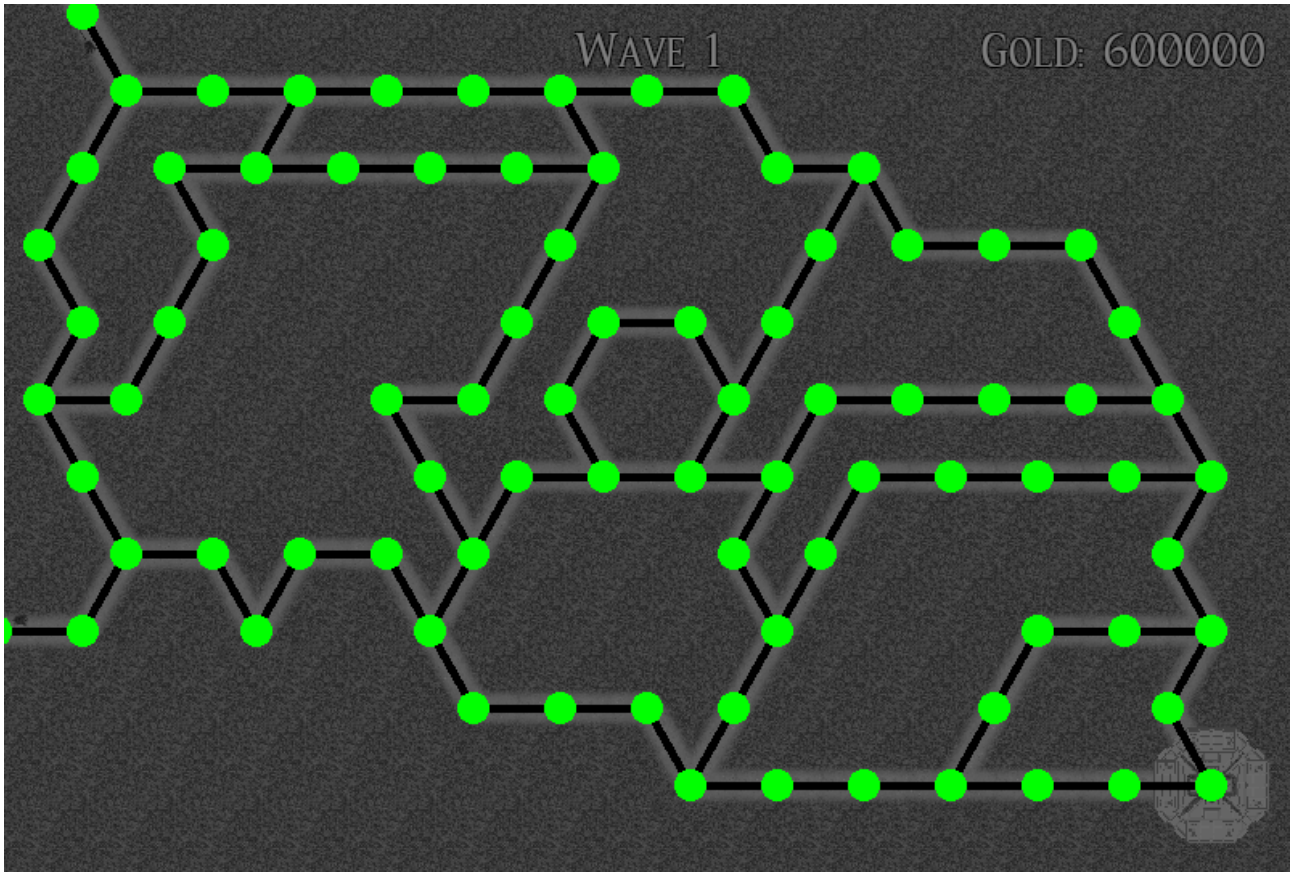

Figura 6.1: Grafo construído após a geração do mapa

6.2 Memória *versus* tempo

Durante a partida, o peso dos nós é modificado. Sempre que uma torre é equipada com um cristal, todos os nós que estão na área de alcance dessa torre têm o seu peso atualizado de acordo com o tipo e a cor do cristal. Porém, a primeira dificuldade que surge é o fato de que cada *creep* possui uma fraqueza e uma resistência diferente, ou seja, o peso dos nós é diferente para cada tipo de *creep*. A segunda dificuldade é que o caminho ótimo

entre os pontos de nascimento dos *creeps* e a base não é fixo durante a partida inteira, pois os pesos dos nós mudam, e portanto o A* deve ser recalculado.

Para a diferença de fraquezas e resistências não há muitas alternativas, e a decisão tomada foi criar um grafo para cada tipo de *creep*. Logo, existem 25 grafos, cada um com pesos diferentes nos nós, e portanto, o recálculo do A* deve ser feito para cada um dos grafos. Já é possível notar que é de extrema importância definir da melhor maneira como e quando o recálculo do A* é feito, caso contrário o código pode se tornar bastante ineficiente. A dinâmica da movimentação dos *creeps* é feita da seguinte forma: quando um *creep* chega ao centro de uma célula, ele requisita o próximo ponto na qual ele deve se mover para. Em média, o intervalo entre uma requisição e outra é aproximadamente 3 segundos, mas esse valor depende da velocidade do *creep*, ou seja, pode variar entre 2 e 5 segundos.

Uma solução seria recalcular o A* todas as vezes que um *creep* chega ao centro de uma célula, para saber qual o caminho ótimo da posição atual até a base. Porém, essa implementação é muito ineficiente, pois pode haver muitas dezenas de *creeps* simultaneamente no mapa, e o recálculo é feito sempre, mesmo que não haja atualização dos pesos dos grafos. Pode ser considerado então que o caminho ótimo para cada *creep* é calculado e armazenado no instante do seu nascimento, e recalculado sempre que haja uma atualização dos pesos dos grafos, mas isso significaria que durante o período de um *frame*, haveria o cálculo de 50 instâncias diferentes do A*, pois cada grafo possui 2 pontos de nascimento de *creeps,* e portanto são gerados dois caminhos por grafo. Além disso, essa implementação não produziria o resultado esperado, pois os *creeps* não se comportariam de maneira dinâmica durante a partida, eles teriam caminhos fixos e não responderiam de imediato às ações do jogador.

Em suma, não é possível criar uma inteligência artificial como a proposta que seja simultaneamente muito eficiente e utilize pouca memória. É preciso escolher um dos lados e verificar se os resultados são satisfatórios. Se o problema for tratado de uma maneira muito simples, essa troca entre memória e tempo será em favor da memória, e a eficiência será comprometida [Millington, 2009]. Por ser um jogo, a estratégia implementada focou desde o início em gerar um código bastante eficiente, e para isso utilizou uma grande quantidade de memória para armazenar informações que serão reutilizadas posteriormente.

6.3 Armazenamento dos caminhos ótimos

Quando o grafo com os pesos iniciais é concluído, ele é copiado para um dicionário, cuja chave é o nome de cada *creep* e o valor é o grafo respectivo a cada tipo. É criado também um dicionário com a mesma chave cujo valor é uma lista com dois caminhos, inicialmente vazios. O próposito de uso é que cada *creep* tem acesso aos valores desses dicionários pela chave, e especificamente o dicionário de caminhos pode ser utilizado para obter o caminho correspondente ao ponto de nascimento do *creep.* Por causa do intervalo

de alguns segundos entre as requisições de movimento de cada *creep*, se torna muito interessante que o cálculo do A* seja feito somente quando for necessário.

Logo, quando os primeiros *creeps* surgem no mapa, o caminho que eles requisitam está vazio, e dessa forma é gerado o caminho ótimo para eles. Desde esse momento, já é realizado o armazenamento desses caminhos nos dicionários, e assim todos os *creeps* que correspondem ao mesmo tipo e que nasceram no mesmo ponto terão seus caminhos prontos e não será necessário recalcular o A*. É interessante notar que nessa estratégia, no máximo 5 valores são acessados e modificados no dicionário durante uma onda devido à progressão de dificuldade do jogo, que disponibiliza o surgimento de no máximo 5 tipos de *creeps* a cada onda por motivos de balanceamento, e esse número é ainda menor no início das partidas.

6.4 Atualização dos pesos dos grafos

Quando o jogador equipa uma torre com um cristal, os nós dos grafos que estão no alcance dessa torre têm seus pesos atualizados [Bourg, 2004]. O valor resultante é a soma do peso atual e da quantidade de dano por projétil correspondente ao cristal multiplicado por 10. O motivo da multiplicação é que, idealmente, os nós que não têm nenhuma torre equipada por perto deveriam ter peso zero, pois o custo de movimentação de um *creep* até esse nó é nulo, afinal ele não irá receber nenhum dano. Mas por motivos de implementação, esses nós têm peso 1, e como a quantidade de dano do cristal mais fraco é 10, seria possível que, por exemplo, um caminho ótimo calculado pelo A* passasse por um nó dentro da área de alcance de uma torre equipada com um cristal ao invés de passar por um caminho longo (com mais de 10 nós) que não tivesse nenhuma torre equipada por perto.

O caminho gerado não seria ótimo, pois apesar de ser um caminho curto, o *creep* sofreria dano, e nesse caso haveria uma alternativa em que o *creep* não sofreria dano algum, que deveria ser o caminho gerado pelo A*. Logo, essa decisão impede esse erro, pois o peso mínimo passa de 11 para 101, e não é possível que um caminho tenha mais de 100 células devido às restrições de tamanho do mapa. Além disso, durante a atualização dos pesos, é verificado se o grafo corresponde à um tipo de *creep* cuja fraqueza ou resistência é igual à cor do cristal. Se a cor do cristal for igual à fraqueza, o peso adicional do nó é dobrado, e se for igual à resistência, o peso é reduzido pela metade. Dessa forma, os *creeps* se comportam de maneira coerente com o dano real que recebem dos cristais.

A consequência que os pesos diferentes entre os grafos gera é muito interessante, pois durante uma partida é possível observar que, dada uma determinada configuração de torres e cristais, cada tipo de *creep* possui seu próprio caminho ótimo até a base [Millington, 2009]. É importante notar que, apesar da atualização dos pesos ser realizada nos 25 grafos, essa operação é muito rápida e pouco custosa computacionalmente. Após a atualização dos grafos, todos os caminhos armazenados no dicionário são apagados, pois eles não correspondem mais aos caminhos ótimos. Seguindo a mesma lógica, os caminhos

cujas cópias são mantidas pelos *creeps* também são apagados. As funções a seguir tratam da atualização dos grafos:

```
func update_graph_weights(tower, gem_dmg, gem_color):
    gem_dmg *= 10
    for cell in tilemap.get_used_cells():
        if tilemap.get_cellv(cell) != ts_db.GRASS:
            var cell_pos = tilemap.map_to_world(cell) + offset
            if cell_pos.distance_to(tower.position / self.scale.x) < \
                    tower.radius / self.scale.x:
                a_star.update_weight(idx_dict[cell_pos], gem_dmg, gem_color)
    a_star.reset_spawn_paths(self)
```

```
func update_weight(cell_id, gem_dmg, gem_color):
    var weight
    for key in graphs.keys():
        weight = graphs[key].get_point_weight_scale(cell_id)
        if creep_info.get_creep_weakness(key) == gem_color:
            gem_dmg *= 2
        elif creep_info.get_creep_strength(key) == gem_color:
            gem_dmg = float(gem_dmg) / 2
        graphs[key].set_point_weight_scale(cell_id, weight + gem_dmg)
```

Quando os *creeps* requisitarem um ponto para se mover, será identificado que os caminhos estão vazios. Logo, o A* irá gerar os caminhos e esses serão armazenados no dicionário. Porém, durante a partida em andamento, é comum que ocorra o caso em que um *creep* está em um ponto que pertencia ao caminho anterior e não pertence mais ao caminho atual. Quando esse fato ocorre, é calculado o caminho entre o ponto atual do *creep* e a base, e armazenado no próprio *creep*. Esse caminho auxiliar também é apagado quando os pesos dos grafos são atualizados e quando o *creep* volta para o caminho armazenado no dicionário, caso haja alguma intersecção com o caminho auxiliar. As funções a seguir devolvem um ponto requisitado por um *creep*.

```
func get_next_point(pos, creep):
    if creep.spawn_path.size() == 0:
        creep.spawn_path = map.update_path(creep.spawn_graph, \
                creep.spawn_path, creep.spawner.position)
        map.a_star.set_spawn_path(map, creep)
    if pos in creep.spawn_path:
        creep.path = null
        return find(pos, creep.spawn_path)
    if creep.path == null:
        creep.path = map.update_path(creep.spawn_graph, \
                creep.path, pos)
    return find(pos, creep.path)

func find(pos, path):
    for i in range(path.size()):
        if pos == path[i]:
            return path[i + 1]
```

A figura a seguir exibe um grafo com pesos, sem influência de fraquezas e resistências de *creeps*. A cor dos nós representam o valor numérico do peso. Nós com peso baixo tem um tom mais próximo de azul, peso intermediário mais próximo de verde, e peso alto mais próximo de vermelho.

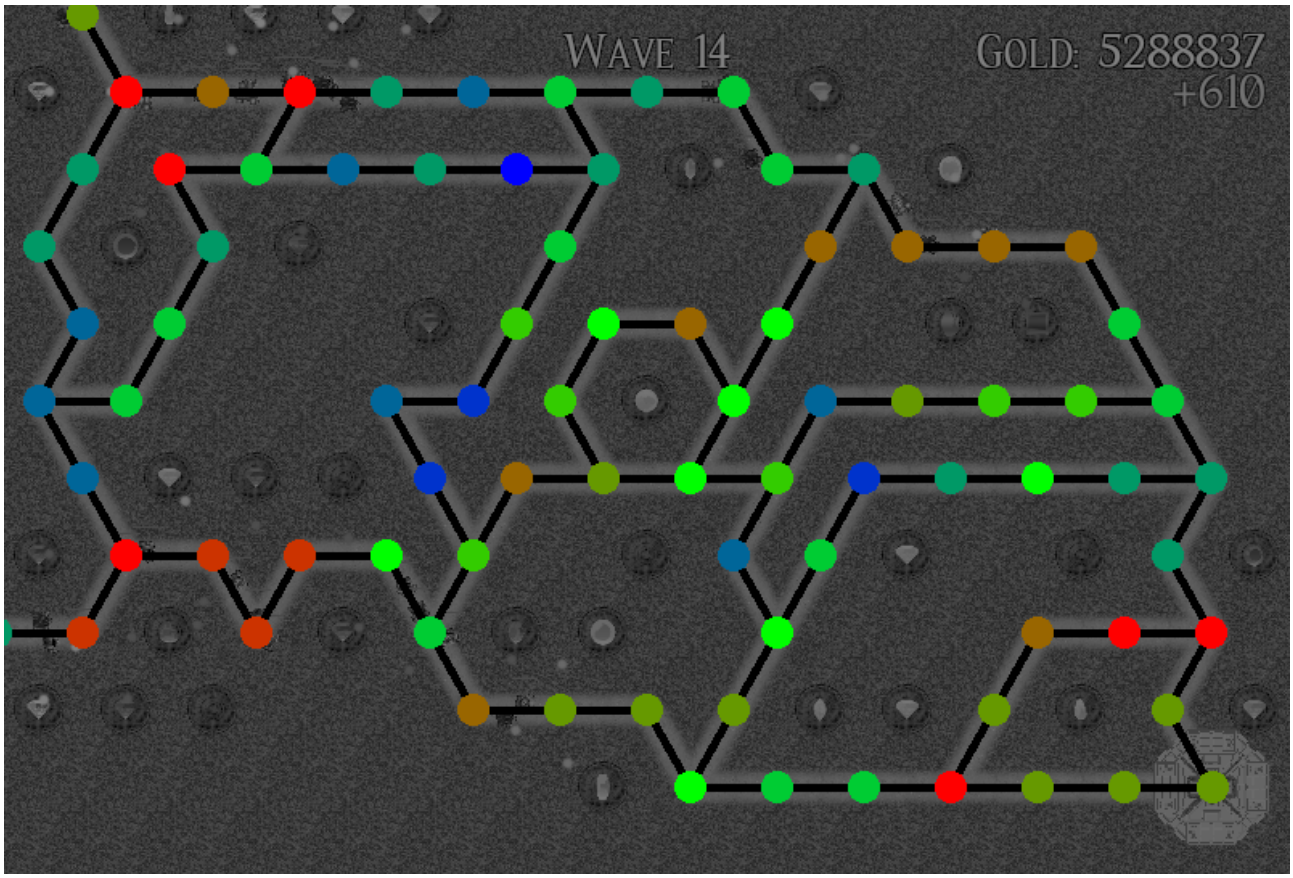

Figura 6.2: Grafo ponderado durante uma partida

Capítulo 7

Resultado e Conclusão

7.1 Balanceamento

Após a conclusão do desenvolvimento, uma das etapas importantes é balancear o jogo. O balanceamento permite que o jogador tenha uma boa experiência durante a jogatina, pois mantém um nível de progressão de dificuldade justo, apresentando desafios durante as várias etapas de uma partida. Também evita que o jogo seja muito fácil, gerando desinteresse ao jogador, ou o contrário, acarretando em frustração [Schell, 2015]. Em *Path of the Wicked*, alguns valores relevantes para o balanceamento são: o preço e o dano de cada tipo de cristal, o preço inicial e os preços subsequentes das torres, o intervalo de tempo entre cada tiro efetuado por um cristal, a velocidade dos projéteis, a quantidade de dinheiro inicial, a recompensa em dinheiro ao abater os *creeps*, o valor da punição quando um *creep* atinge a base, a quantidade de pontos da primeira onda e a progressão das demais ondas, o intervalo de tempo entre o fim de uma onda e o início da próxima, o intervalo de tempo entre o surgimento de um *creep* e outro. Mas, sem dúvidas, os valores mais importantes que realmente definem a progressão de dificuldade são os atributos dos *creeps*.

Como definido anteriormente, o valor de um *creep* é proporcional à sua velocidade multiplicado pelos seus pontos de vida, e indica o nível de dificuldade correspondente para abatê-lo. Por motivos visuais, as velocidades dos *creeps* foram definidas manualmente, baseadas na aparência de cada um, e variam entre 20 e 60. Logo, é necessário definir a progressão dos valores, e dessa forma obter os pontos de vida de cada *creep.* Essa etapa foi composta de várias iterações, e se mostrou bastante desafiadora. O resultado final foi:

$$v(i) = f(i) + q(i)$$

em que $v(i)$ é o valor do *creep* indexado em i , $0 \leq i \leq 24$. $f(i)$ é definido por:

$$f\ (i) = \begin{cases} 377, & se\ i = 0 \\ 610, & se\ i = 1 \\ f\ (i-2) + f\ (i-1)\ , & se\ 2 \leq i \leq 5 \\ f\ (i-2) + f\ (i-1)/2, & se\ 6 \leq i \leq 17 \\ f\ (i-2) + f\ (i-1)/3, & se\ 18 \leq i \leq 21 \\ f\ (i-2) + f\ (i-1)/4, & se\ i \geq 22 \end{cases}$$

$q(i)$ definido por:

$$q\ (i) = min\ (b\ (i)^2\ ,\ 20000)$$

e $b(i)$ definido por:

$$b\ (i) = \begin{cases} 43, & se\ i = 0 \\ b\ (i-1) + min\ (i+4,\ 10)\ , & se\ i > 0 \end{cases}$$

As imagens a seguir mostram o gráfico das funções $v(i)$ (em verde), $f(i)$ (em azul) e $q(i)$ (em laranja). O primeiro gráfico apresenta o eixo y em escala linear, e o segundo, em escala logarítmica.

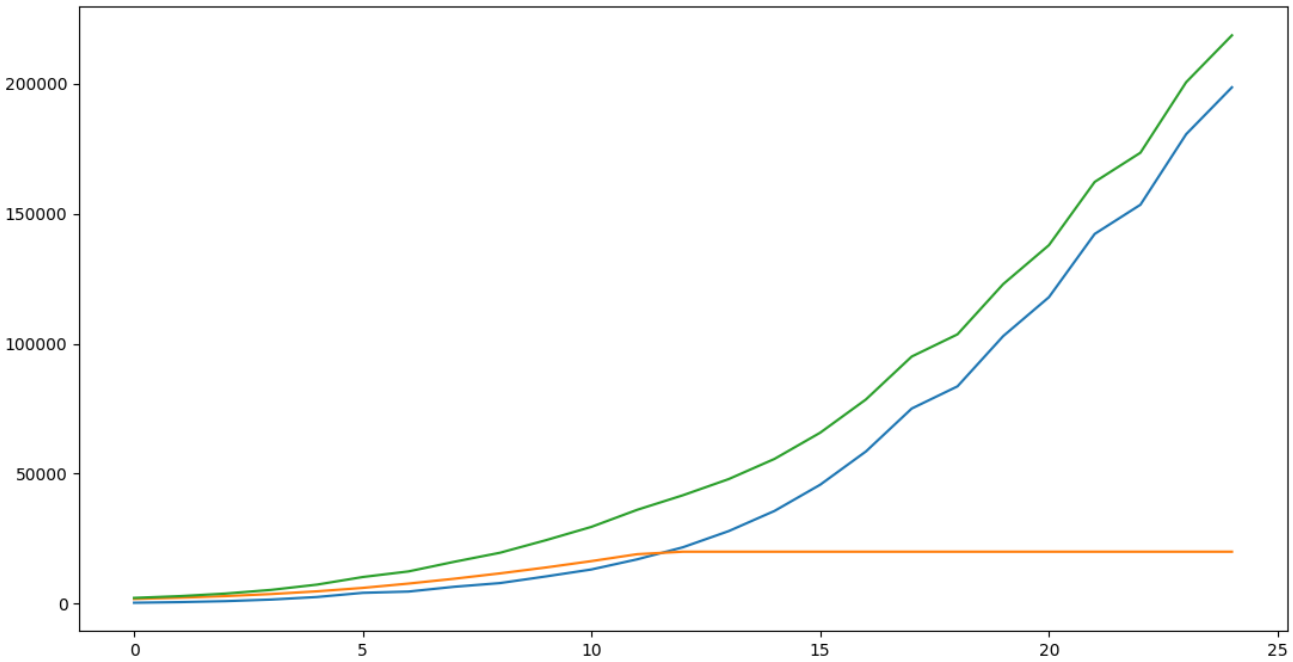

Figura 7.1: Gráfico exibindo a progressão de valores dos *creeps* em escala linear

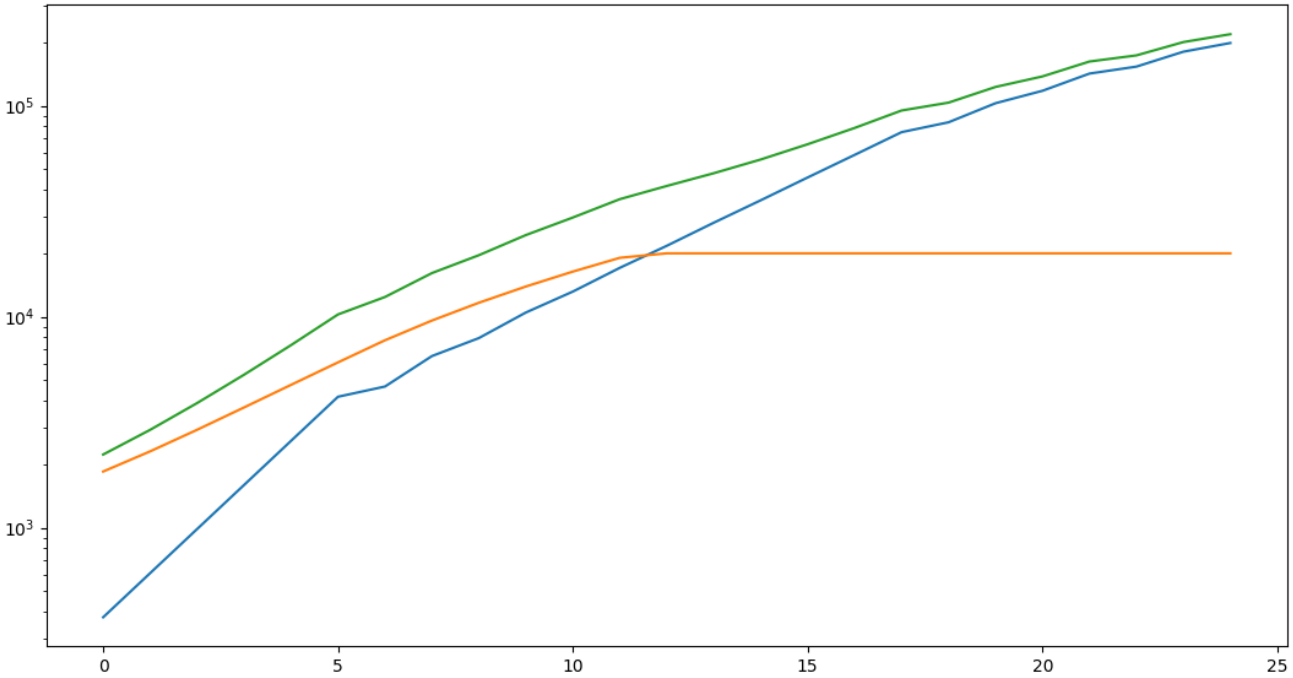

Figura 7.2: Gráfico exibindo a progressão de valores dos *creeps* em escala logarítmica

7.2 Considerações finais

Desenvolver o *Path of the Wicked* do zero foi uma ótima forma de estudar sobre geração procedural, aplicar algoritmos muito importantes da área de Inteligência Artificial, e exercitar conceitos de *game design*, sempre com foco na eficiência do código. O resultado foi um jogo divertido e desafiador, que proporciona uma alta interação com o jogador através do comportamento dinâmico dos *creeps*, pois constantemente dá margem para otimizações na configuração de torres e cristais.

Referências Bibliográficas

[**Yannakakis, 2018**] YANNAKAKIS, G.; TOGELIUS, J. *Artificial Intelligence and Games*. 1 ed. Springer, 2018. p. 184, 185. 1, 34

[**Short, 2017**] SHORT, T.; ADAMS, T. *Procedural generation in game design*. 1 ed. Boca Raton: Taylor & Francis, CRC Press, 2017. p. 134, 275-277, 287. 1, 31, 34

[**Schell, 2015**] SCHELL, J. *The Art of Game Design: A Book of Lenses*. 2 ed. Boca Raton: Taylor & Francis, CRC Press, 2015. p. 207, 223. 3, 43

[**Avery, 2011**] AVERY, P. et al. *Computational Intelligence and Tower Defence Games*. 2011 IEEE Congress of Evolutionary Computation (CEC). Nova Orleans: IEEE, 2011. p. 2-4. 3

[**Nystrom, 2014**] NYSTROM, R. *Game Programming Patterns*. 1 ed. Genever Benning, 2014. p. 123-139. 6

[**Linietsky, 2018**] LINIETSKY, J.; MANZUR, A. *GDScript*. http://docs.godotengine.org/en/stable/getting_started/scripting/gdscript/gdscript_basics.html. 2018. Acesso em: 15 out. 2018. 11

[**Patel, 2018**] PATEL, A. *Hexagonal Grids*. 2018. https://www.redblobgames.com/grids/hexagons/. Acesso em: 18 nov. 2018. 30

[**Millington, 2009**] MILLINGTON, I.; FUNGE, J. *Artificial Intelligence for Games*. 2 ed. Burlington: Morgan Kaufmann Publishers, 2009. p. 25-31, 197, 198, 216, 239, 240, 553-559. 31, 37, 39, 40

[**Bourg, 2004**] BOURG, D.; SEEMAN, G. *AI for Game Developers: Creating Intelligent Behavior in Games*. 1 ed. O' Reilly Media, 2004. p. 176, 194-200. 37, 40