PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Labirinto

Geração Procedimental de níveis
de Dungeon TDML e TDCL

Thiago Melcop Sant'Anna

PROJETO FINAL DE GRADUAÇÃO

Centro Técnico Científico – CTC
Departamento de Informática

Curso de Graduação de Ciência da Computação

Rio de Janeiro, Novembro de 2023

Thiago Melcop Sant'Anna

Labirinto

Geração Procedimental de níveis de Dungeon TDML e TDCL

Relatório de Projeto Final, apresentado ao programa de Projeto Final da PUC-Rio como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Orientador: Augusto César Espíndola Baffa

Rio de Janeiro

Novembro de 2023

Resumo

Thiago Melcop Sant'Anna; Augusto César Espíndola Baffa; **Labirinto, Geração Procedimental de níveis de Dungeon TDML e TDCL**. Rio de Janeiro, 2023. 30p. Projeto de Graduação – Departamento de CENTRO TÉCINICO CIENTÍFICO – CTC DEPARTAMENTO DE INFORMÁTICA, Pontifícia Universidade Católica do Rio de Janeiro.

O projeto consiste no desenvolvimento de um jogo ("Labirinto") onde será demonstrado a geração procedimental de níveis utilizando o ambiente Monogame na linguagem C#. A geração procedimental foca na união e utilização de duas categorias distintas de criação de dungeons, uma focada em quartos conectados por corredores e outra ondes quartos são conectados diretamente por paredes e passagens. O projeto foi desenvolvido com o intuito de servir como uma demonstração prática do uso de geração procedimental e utilização de conceitos já existentes e utilizados por desenvolvedores e designers de jogos.

Palavras-chave

C#, Desenvolvimento de Jogos, Geração Procedimental, Monogame.

Abstract

Thiago Melcop Sant'Anna; Augusto César Espíndola Baffa; **Labyrinth, Procedural Level Generation of TDML and TDCL Dungeons**. Rio de Janeiro, 2023. 30p. Final Project – Departamento de CENTRO TÉCINICO CIENTÍFICO – CTC DEPARTAMENTO DE INFORMÁTICA, Pontifícia Universidade Católica do Rio de Janeiro.

The project consists of the development of a game ("Labirinto") where procedural generation of levels will be demonstrated using the Monogame environment on the C# language. The procedural generation focuses on the union and utilizations of two categories on the generation of dungeons where one is focused on the rooms connected to each other by corridors and the other on rooms connected directly by walls and gaps in those walls. The project was developed with the desire to be a practical demonstration of procedural generation and the utilization of already existing concepts that are used by game developers and designers.

Keywords

C#, Game Development, Procedural Generation, Monogame.

Conteúdos

1. Introdução

Jogos e geração procedimental tem uma história longa e duradoura, um exemplo clássico e que por décadas define um genro inteiro de jogos é Rogue(1980), de onde o nome Rogue-Like ou Rogue-Lite é derivado. Esse gênero de jogo é marcado primeiramente com a ideal onde o jogador irá tentar completar o jogo múltiplas vezes, onde a morte do personagem leva ao jogo a acabar e obriga o jogador a retornar ao começo[17].

Em Rogue e jogos que carregam seu legado, é comum que em cada tentativa novos nível ou sequência de novos níveis sejam gerados completamente diferentes de quaisquer níveis que foram jogados anteriormente, fazendo que cada tentativa seja única e que não é possível ser bom em um nível qualquer, mas que o jogador tenha que aprender com os fundamentos do jogo funciona para ter uma maior chance de sucesso[17].

Para realizar essa funcionalidade de cada tentativa ser única, jogo como Rogue utilização de geração procedimental usando parâmetros e heurísticas que garantem que cada nível não é apenas único, mas também é capaz de ser concluído dentro das regras, limitações, e ferramentas oferecidas pelo jogo.

Em muitos jogos, e principalmente em Rogue, os níveis são imaginados como “masmorras” ou *dungeons* como serão referidos de agora em diante, consiste em diversos quartos. *Dungeons* e seus quarto claustrofóbicos formam um ambiente perfeito para geração procedimental devido ao seu número mais limitado de possibilidades em relação a um mundo aberto.

Dungeons e geração procedimental são um par comum em diversos jogos modernos, por exemplo Binding of Isaac Rebirth(2014), Bloodborne(2015), Crypt of the NecroDancer(2015), Enter the Gungeon(2016) e Loop Hero(2021), todos esses, menos Bloodborne, são também Rogue-likes.

Dungeons, de acordo com Viana[1], podem ser dívidas em 3 categorias: “top-down mansion-like dungeon” (TDML), “top-down cavern-like dungeon” (TDCL) e “side-scrolling dungeon” (SS). Ambos TDML e TDCL consistem em dungeons vista “por cima” ou uma secção horizontal, e SS é comparação é uma vista “de lado” ou uma secção vertical.

TDML consiste em uma dungeon onde quarto são diretamente conectados uns aos outros sem a necessidade de corredores, dois um mais quartos dividem as mesmas paredes como divisória entre eles[1]. Jogos característicos desse tipo de dungeon são a dungeons clássicas de Legend of Zelda(1986) e Binding of

Isaac Rebirth(2014), onde para acessar outros quartos é apenas necessário atravessar uma porta ou passagem entre os dois.

Nesses casos mencionados, essas dungeons possuem telas que estão dividem cada quarto, o jogador é incapaz de observar qualquer aspecto de um quarto adjacente ou o quarto está preste a entrar até atravessar a porta ou vão. Isso não necessariamente verdadeiro de todas as *dungeons* TDML, mas é um aspecto comum, pois cada quarto e os inimigos e outro fatores podem apenas ser processado quando necessário: A presença do jogador dentro do espaço do quarto.

Porém, apenas a ausência de corredores e a presença quartos que conectam uns aos outros diretamente seria necessário para a classificação desses como TDML.

Em TDCL os quartos são conectados por corredores, esses corredores podem ser variados, desde corredores retos que percorrem o menor caminho entre os dois quartos a aqueles gerados para ter um aspecto cavernoso e não-uniforme. Jogos exemplares desse tipo de geração de dungeon são Pokémon Mystery Dungeon: Explorers of Time(2007) e Rogue.

SS em vários aspectos diverge do outro dois, ainda é composto de quartos conectados, porem sua perspectiva lateral leva a outras consideração na geração de mapas, incluindo um necessidade de lidar com a verticalidade do mapa, portanto a geração do próprio quarto deve garantir que as saídas possam ser acessadas com a ferramentas oferecidas ao jogador, exemplos onde essas dungeons podem ser encontradas são em Super Metroid(1994), onde não ocorre geração procedimental, porém ainda é um exemplo claro do aspecto que dungeons possuem, Dead Cells(2018) onde ocorre geração procedimental em um aspecto fundamental do jogo.

Para este projeto os tipos de *dungeon* relevantes são TDML e TDCL. Nesse projeto a intenção é a criação de um algoritmo capaz de procedimentalmente criar ambas *dungeons* independentemente e juntar ambas as categorias dungeons em um nível. Esses níveis serão explorados dentro de um contexto de um jogo, testando de forma natural os limites dessa geração procedimental.

Para uma possível integração com estilo SS, seria necessário um pensamento e design diferente, pois para alternar do meio *Top-Down* para um *Side-scroller* seriam necessárias muitas mudanças para permitir essa funcionalidade e coerência de design, seria um escopo muito maior para integrar esses.

O jogo Labirinto foi feito utilizando o Monogame[11], uma *Engine Open-Source* que utiliza a linguagem C#, uma linguagem muito comum no desenvolvimento de jogos, incluindo a plataforma Unity[13]. O próprio MonoGame é uma *Engine* utilizada por diversos desenvolvedores, principalmente desenvolvedores indie, alguns exemplos de jogos feitos em MonoGame são: Barotrauma(2023), Celeste(2018), Fez(2013), Pyre(2017) e Stardew Valley(2016).

O ambiente utilizado para o desenvolvimento, teste e execução do programa em Windows 10.

2. Situação Atual

Na indústria moderna de jogos, geração procedimental é usada quase onipresentemente, empresas independentes de menor escala e empresas enormes ambas utilizam da geração procedimental para ampliar e expandir a quantidade de conteúdo disponível no jogo com uma quantidade de trabalho menor pelos desenvolvedores e designers.

Como explicado por Tutenel[10], com a expansão do escopo desses mundos virtuais e sua complexidade, existe a necessidade de criar algoritmos que possam gerar conteúdo para popular esses mundos. Muitas empresas passam a utilizar formas hibridas de conteúdo, misturando conteúdo feito-a-mão com conteúdo procedimentalmente como foi feito pela Bethesda mais recentemente em Starfield(2023).

A indústria de jogos sempre está a produzir novos algoritmos e programas para a criação procedimental de conteúdo, principalmente desenvolvedores com menor quantidade de recurso[3], pois nesses grupos a quantidade de tempo e esforços que podem ser dedicados a conteúdo feito-a-mão é menor.

Técnicas e algoritmos de geração procedimental, por mais simples ou complexos adicionam ao conhecimento coletivo, podendo servir de base ou ser expandido um projeto atuais ou futuros.

Este projeto tem como base a demonstração a criação de um algoritmo de geração procedimental, em particular a criação de *dungeons* geradas procedimentalmente, e como mencionado anteriormente na introdução, muitos jogos clássicos e modernos ainda utilizam de dungeons como o fundamento de seus níveis.

O algoritmo descrito por Sampaio[2], descreve uma forma eficiente de geração procedimental e foi entre algumas das inspirações para a criação de dungeons no estilo TDML dentro do projeto, outra inspiração e influência foram os algoritmos propostos por Van Der Linden[7].

Os algoritmos que foram usados como inspiração para a geração de dungeons no estilo TDCL um foi o utilizado em RogueSharp[12] onde Rogue é recriado em C# e o outros apresentados por Van Der Linden[7].

3. Objetivos

Objetivos Gerais

O projeto tem como objetivo a criação de um jogo onde para completar progredir o jogador deve explorar e sobreviver em um labirinto, uma *dungeon*.

Essa *dungeon* será criada a partir de um algoritmo que é capaz de gerar *dungeons* dentro das categorias TDML e TDCL, essa *dungeon* iria conter perigos em forma de inimigos que perseguem o jogador e escadas que levam a níveis mais baixos do labirinto onde jogo ficaria mais difícil. Todos os níveis seriam gerados de forma procedimental.

Exploração do nível

O jogador deverá ser capaz de explorar a *dungeon* utilizando a setas do teclado. Para progredir os níveis o jogador deverá encontrar a cada nível uma quantidade de ouro mínima antes depois descer uma escada para o próximo nível.

A exploração é uma parte importante do jogo, portanto ao entrar em um nível o mapa não é inicialmente visível pelo jogador além dos quadrados que ele pode ver na sua proximidade, tudo que o jogador explora e vê é adicionado a “memória” que excederia a sua visão, facilitando navegação futura.

Geração de Dungeon e Níveis

As *dungeons* geradas devem seguir as categorias definidas anteriormente de TDML e TDCL. Para isso, zonas ondes os quartos dentro delas serão construídos e conectados pertencendo a uma das categorias.

Todos os quartos gerados em cada nível devem obrigatoriamente ter a capacidade de ser acessados e explorados, todo nível gerado deve contém ouro com valor suficiente para que, no pior caso, o jogador possa progredir para o próximo nível, todo nível de gerar pelo menos uma escada para que o jogador possa passar para o próximo nível.

A geração de *dungeon* e todos os níveis e a localização e valor do ouro contidos dentro dele devem ser os mesmos baseados da mesma semente, portanto a geração de níveis deverá conter um gerador de números independente de qualquer outros que seriam utilizados pelo jogador ou por inimigos.

Mais uma qualidade que é esperada da geração de *dungeons* é a velocidade da geração, que não deve durar mais que alguns instantes para gerar o nível, particularmente é níveis mais baixos onde o tamanho e número de quartos e zonas é significantemente menor do que é níveis mais altos.

Inimigos

O jogo deverá conter inimigos para dar ao jogador uma sensação de urgência e perigo e permitir um estado de “derrota” seja uma possibilidade. Os inimigos iram perseguir o jogador quando os dois estiverem próximos, obrigando o jogador a escolher o seu caminho mais cuidadosamente.

Inimigos obviamente não poderiam aparecer no mesmo espaço que o jogador, para dar tempo ao jogador para reagir, os inimigos também devem ao estarem distantes do jogador, não o perseguir.

Os inimigos também devem ser capazes de ter o aspecto de serem inteligentes, podendo investigar a ultima posição que viram o jogador, antes de desistirem de procurá-lo.

Níveis e Dificuldade

A cada nível que o jogador progride o mapa ficará maior com menos escadas em relação ao número de quartos e zonas, e os inimigos apareceram em maior volume, serão mais rápidos, e mais frequentemente buscariam a posição do jogador.

Juntamente, a quantidade de ouro necessária para acessar as escadas aumentaria a cada nível, obrigando o jogador mesmo tendo sorte em achar uma escada imediatamente, forçá-lo a procurar mais ouro antes de conseguir progredir.

Porém o jogador também poderia acumular ouro de níveis anteriores para precisar pegar menos ou nenhum ouro em certo nível. Porém cada nível, obrigatoriamente, precisaria ter valor mínimo suficiente para passar o jogo para o próximo nível.

4. Atividades Realizadas

Estudos Preliminares

A *Engine* MonoGame utiliza a linguagem C#, uma linguagem que não tinha uma grande familiaridade com antes do início do projeto, porém tinha pratica com diversas linguagens similares como C, C++, e Java, portanto passar a utilizar C# não levou tempo considerável, apenas foi necessário aprender onde ela se

diferenciava das linguagens similares, pois os conceitos básicos e fundamentos de uma linguagem orientada a objeto são eram bem conhecidos.

Também havia uma pequena familiaridade com desenvolvimento de jogos em Unity que utiliza a linguagem C#, porém até antes deste projeto nunca havia embarcado em um projeto sério para o desenvolvimento de um jogo.

Estudos conceituais e de tecnologia

Para fazer este projeto foi necessário estudar o funcionamento do *Engine* MonoGame e me familiarizar como funciona o fluxo da *pipeline* de criação de conteúdo e do jogo dentro da *Engine*.

O MonoGame é uma *Engine Open-Source* para o desenvolvimento de jogos, possui menor bagagem do que Unity, e é relativamente simples em comparação.

Por exemplo em Unity é necessário trabalhar dentro do editor de cena, e utilizar um ambiente 3D mesmo quando realizado um projeto de um escopo mais simples. O ambiente 3D tem suas vantagens, porém não caiam dentro do escopo deste projeto.

MonoGame é utilizado por diversas empresas, particularmente empresas indies, devido à sua simplicidade e o fato de seu código ser *Open-Source*, evitando limitações e custos impostos por *Engine* que precisam ser licenciadas para o uso comercial.

Geração procedimental é um conceito já familiar e que tinha experiência com, e parte foi a motivação para a seleção do tópico desse projeto. Porém foi necessário aprender inúmeros métodos e filosofias por trás de geração procedimental. Em particular o livro de Short, Procedural Generation in Game Design[3] onde desenvolvedores demonstraram e cometeram sobre as suas experiências e ideias de geração procedimental, e a compilação de métodos de geração procedimental de dungeons por Van Der Linden[7] foram instrumentais para o estudo do conceito de geração procedimental e para a criação do projeto.

Uma inspiração para a criação do algoritmo que seria utilizado no projeto foi um apresentado por Sampaio[2], nesse algoritmo uma filosofia importante é não importa muito se os quartos e corredores estão posicionados uns em cimas dos outros, o algoritmo continua a criar e montar os quartos. Na verdade, quartos sobrepostos fazem parte da intenção do algoritmo, permitindo construídos inicialmente quartos simples possam ter tamanhos maiores e silhueta mais complexas pois cada quarto adiciona um em cima do outro, algo que acabou sendo aplicado no Labirinto.

Testes e Protótipos para aprendizado e demonstração

Para aprender MonoGame e C# forma produzidos três pequenos jogos baseados em um curso on-line: Discover Game Development with C# Programming and “MonoGame”[14] dentro da plataforma Udemy. Esses três serviram como protótipo para a criação do jogo.

Esses três jogos formam fundamentais para aprender como desenvolver em MonoGame. O primeiro jogo desenvolvido foi um jogo chamado pelo curso “Shooting Gallery”, onde o objetivo é utilizar o mouse para “atirar” em alvos que aparecem na tela, o jogador tem um total de 10 segundos para clicar em alvos, ao final dos dez segundos a pontuação final é exibida e o jogo é terminado.

Como introdução ao MonoGame esse primeiro jogo cobriu diversos fundamentos do MonoGame, uns do aspectos essenciais desse aprendizado foi a capacidade introduzir imagens ao jogo utilizando o manager de conteúdo do próprio MonoGame, conforme é apresentado na figura 1.

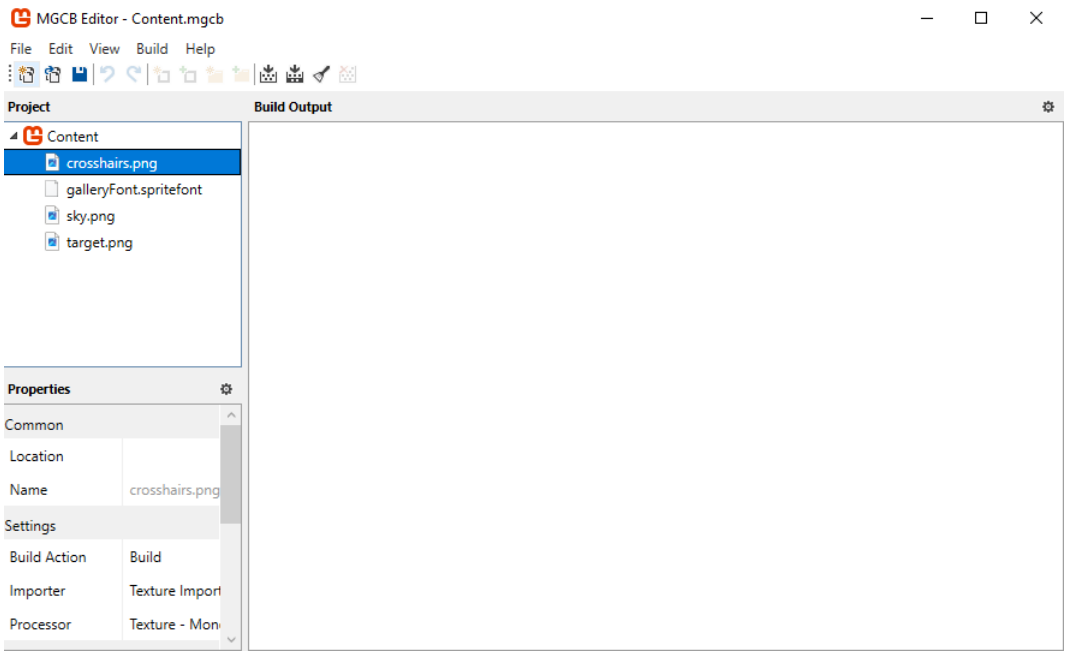

Fig. 1 Janela de edição de conteúdo do MonoGame

E outro foi de como utilizar e operar a pipeline de cada frame, incluindo o update de cada frame e os seus efeitos no estado do jogo, incluindo a detecção de inputs do mouse, e o desenho da janela a cada frame, incluindo o processo de carregar assets. Podemos ver o jogo em progresso nas figuras 2 e 3.

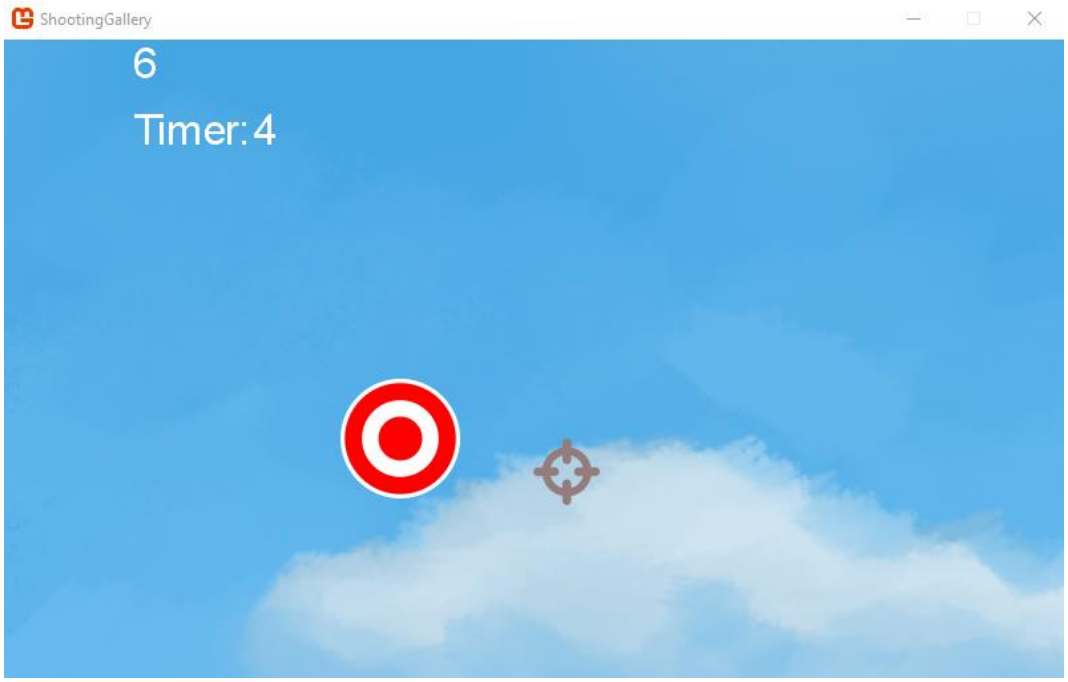

Fig.2 Shooting Gallery em progresso

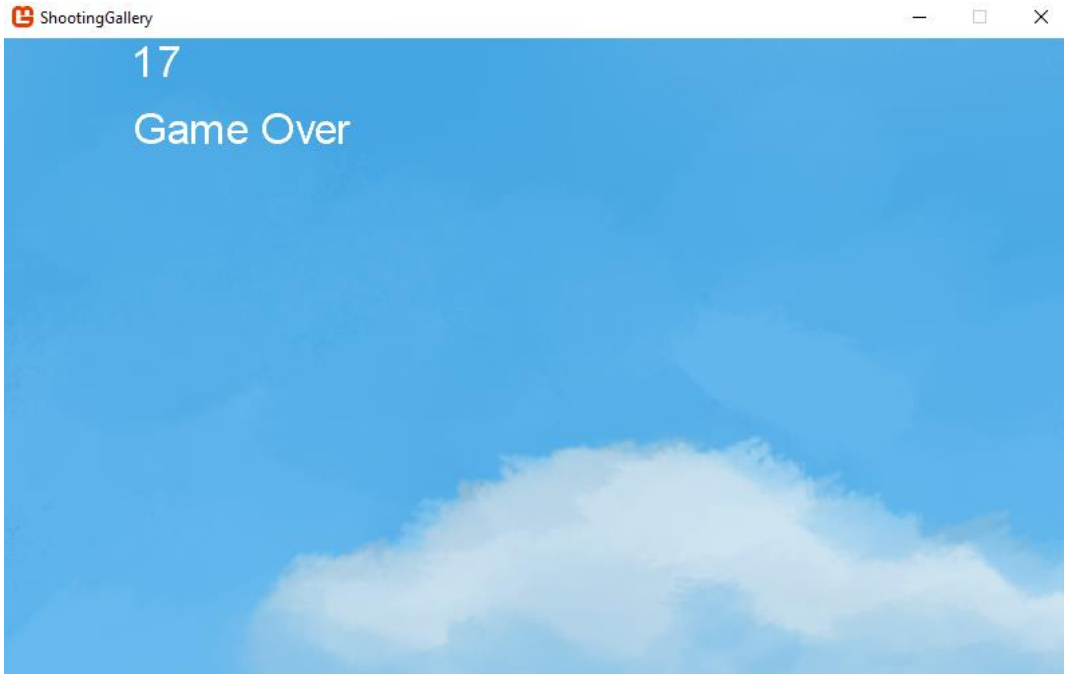

Fig. 3 Shooting Gallery ao acabar o tempo

O segundo jogo feito foi um jogo baseado em usar uma espaçonave (dando o nome ao jogo “Spaceship”) para desviar de asteroides e sobreviver o máximo de tempo possível, o jogo continha uma função que criava asteroides no canto esquerdo da tela e a cada frame os asteroides moviam para o lado esquerdo. Quando mais tempo passava mais vezes a função de criar asteroide era chamada,

deixando sobrevivência mais difícil a cada segundo. O jogo pode ser observado na figura 4.

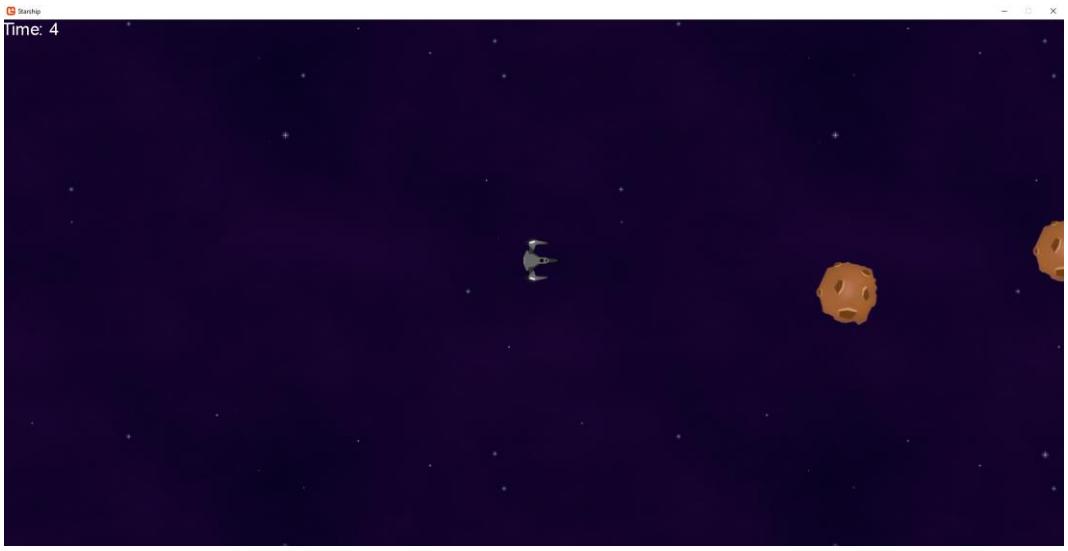

Fig. 4 Spaceship em progresso

Dentro do curso jogo introduziu a utilização de inputs do teclado e a verificação de colisão entre objetos dentro do jogo. Ambos os conceitos seriam utilizados no projeto para explorar o ambiente para a colisão com os adversários.

O terceiro e último jogo foi similarmente um onde o objetivo era sobreviver pelo maior tempo possível, porém ao invés de apenas desviar de obstáculos que apenas percorrem uma direção o jogador era capaz de confrontar esses obstáculos disparando ataques contra eles, esses obstáculos também possuem uma inteligência artificial básica para perseguir o jogador, algo Labirinto também acabaria usando. O curso o chamava de “RPG” devido a perspectiva top-down utilizada, semelhante a jogos clássicos, que pode ser observado figura 5.

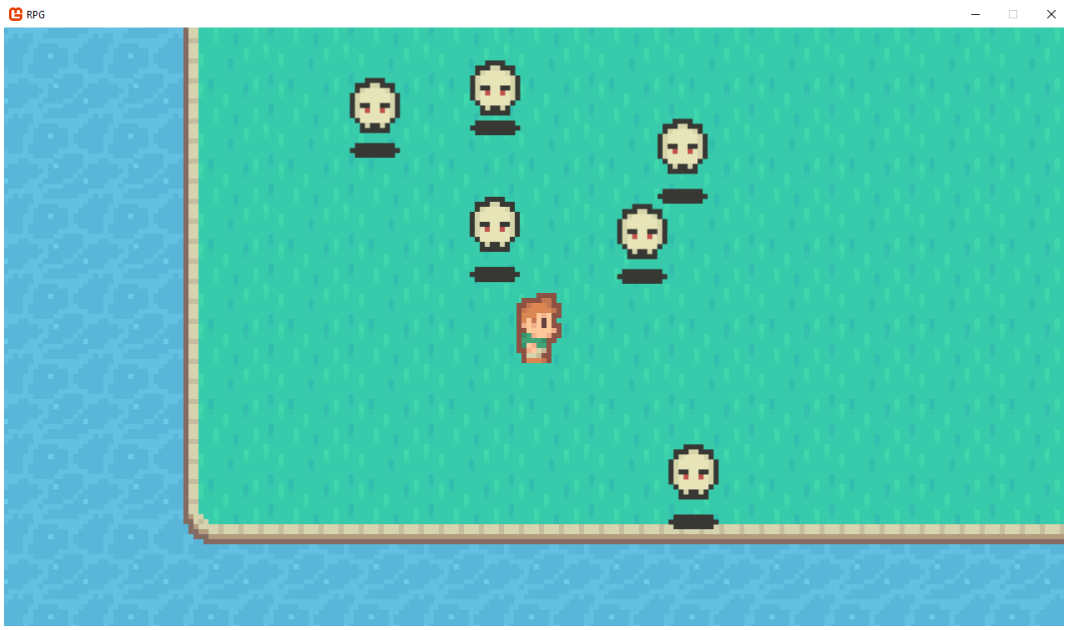

Fig. 5 RPG em progresso

Para esse último jogo também foi introduzido um software O*pen-Source* para o manuseamento da câmera dentro do MonoGame, e dentro do contexto do jogo foi utilizado para que a câmera conseguisse seguir o jogador

Esse software, chamado Comora[15] acabou sendo utilizado no resto do projeto, pois funcionava bem dentro das necessidades e do escopo do projeto.

5. Projetos e especificações do Sistema

Diagramas e Organização

O projeto foi organizado por um diagrama de classe, catalogando como cada elemento do projeto iria se relacionar. O diagrama final de classe (Fig. 6), incluindo todas as alterações realizadas alongo do projeto

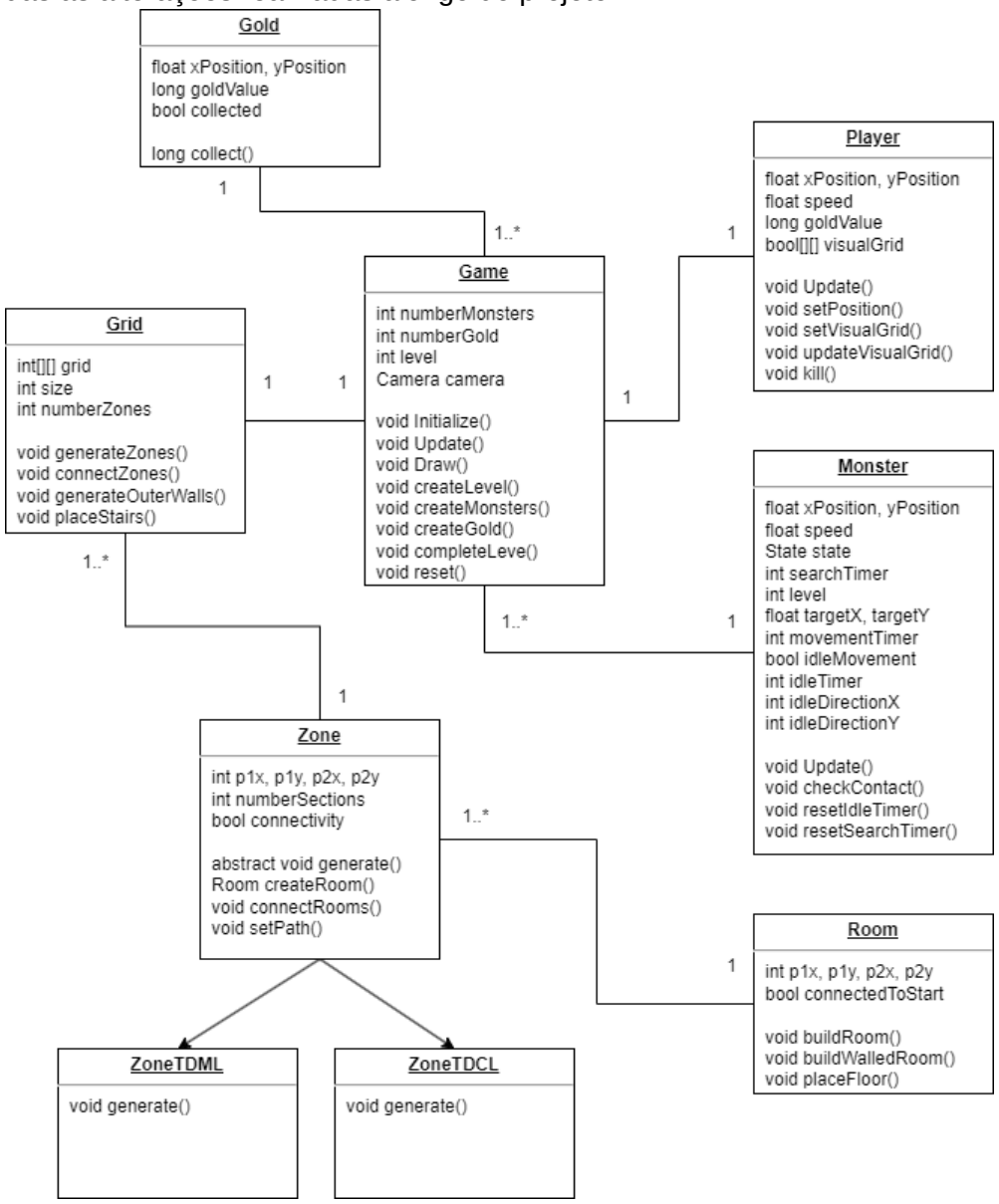

Fig. 6 Diagrama de classe do Labirinto

A classe Game é a classe mestre do jogo, ela coordena informação de todas as outras classes e controlar os dados necessário para passar uma para a outra, incluindo principalmente as classes Player, Monster, Gold, e Grid, ela também contém as funções de inicialização do jogo, o update de cada frame, e do desenho da tela, incluindo a câmera.

A classe Game mantém controle do nível atual que o estado do jogo se encontra, usando esse valor para controlar o tamanho do mapa gerado, a quantidade de entidades a serem geradas.

A classe Game cria uma classe Grid informando o seu tamanho e quantidade de zonas esperadas em relação ao nível atual. A classe Grid cria as zonas (“Zone”) e estabelece que tipo de Zona ele é (TDML ou TDCL).

Cada Zone é responsável por gerar um número de quartos (“Room”), essa geração é baseada no tipo de Zone a qual ela pertence.

Os inimigos dentro do jogo foram criados a partir de uma máquina de estado (Fig. 7), a ideia é básica, mas eficiência, inimigos que não estão perto do jogador estão a perambular, ao avistar o jogador o inimigo tenta se aproximar do jogador diretamente. Se o jogador conseguir escapar do inimigo o inimigo busca até o ponto onde ele viu o jogador pela última vez, se ele chegar a esse ponto, porém não encontrar o jogador, ele volta a perambular, se ele encontrou o jogador, volte a caçar ele.

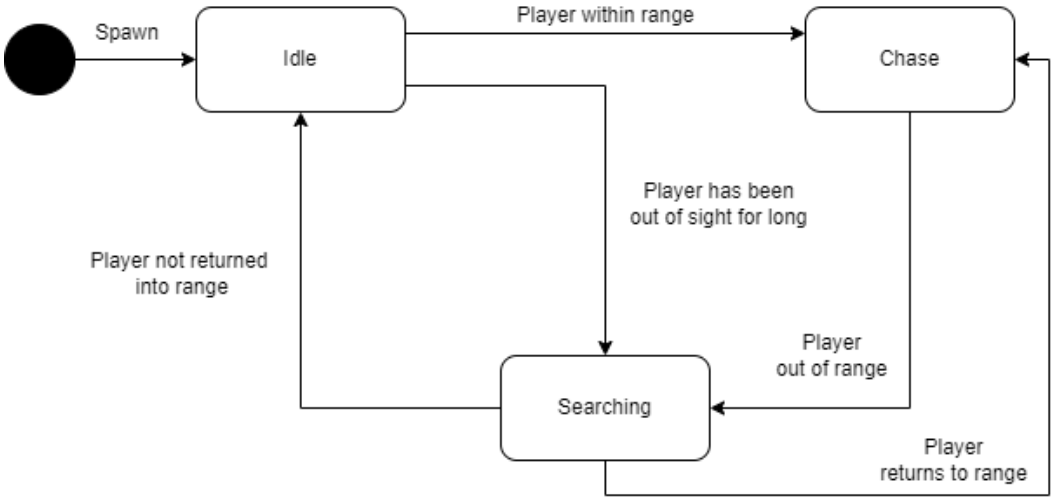

Fig. 7 Máquina de estado dos inimigos

Porém para adicionar dificuldade, e garantir que sempre algum inimigo estará por perto do jogador, se um inimigo estiver a perambular por muito tempo, tempo dependente do nível atual, ele iria buscar uma posição perto do jogador.

Para desenvolver o projeto as seguintes etapas formam determinadas durante o período do Projeto Final I, com pequenas modificações devido a realidade do processo (tabela 1):

1. Criação do documento de Proposta de Projeto Final.
2. O estudo de operação e implementação de algoritmos procedimentais, em particular aqueles com relação a geração de dungeons TDML e TDCL.
3. Estudo da Linguagem C# para o desenvolvimento e implementação do algoritmo.
4. Estudo framework MonoGame para a implementação do jogo e o ambiente para interagir e explorar os níveis procedimentais.
5. A criação do algoritmo de geração procedimental visando a mistura de estruturas TDMLs e TDCLs para a criação de níveis.
6. Implementação do algoritmo dentro do MonoGame
7. Revisão e melhoramento do algoritmo
8. Adicionar complexidade ao algoritmo, em forma de portas, puzzles simples, e níveis, mapas maiores e mais complexos.
9. Criação do Documento Final

Mês/Etapa	Mar		Abr		Mai		Jun		Jul		Ago		Set		Out		Nov		Dez	
1																				
2																				
3																				
4																				
5																				
6																				
7																				
8																				
9																				

Tabela 1: Cronograma do Projeto I

Porém, passando de Projeto Final I para Projeto Final II houve mudanças no processo devido as realidades do processo do desenvolvimento do projeto. Em particular o a criação do algoritmo e o desenvolvimento do jogo no MonoGame foram feitos muitos mais próximas uma da outra devido a facilidade de um ambiente visual para inspecionar e avaliar a gerações possíveis falhas, outro o tempo limite e o nível de complexidade de adicionar complexidade ao algoritmo.

A tabela a seguir (Tabela 2) contém um cronograma revisado do com datas aproximadas do que foi feito durante o Projeto.

Mês/Etapa	Mar		Abr		Mai		Jun		Jul		Ago		Set		Out		Nov		Dez	
1																				
2																				
3																				
4																				
5																				
6																				
7																				
8																				
9																				

Tabela 2: Cronograma revisado

A implementação do algoritmo e a criação conjunta do ambiente de Jogo levou muito mais tempo do que o esperado, portanto não houve a oportunidade de adicionar complexidade. Também tiveram certamente adições, modificações, e problemas que não seriam bem representados aqui.

Módulos Desenvolvidos

Para o desenvolvimento do Labirinto foram criadas diversas Classes que operam uma com as outras para a geração da *dungeon* e para a navegação do jogador.

Explorar o jogo e como funciona a sua geração procedimental de níveis e melhor realizado através das classes que foram criadas para regulamentar esses processos do jogo.

Classe Game

A classe Game serve como o centro da operação do jogo, ela cria e inicia as outras classes e é responsável por lidar com e verificar as mudanças no estado de jogo e executar ou manda outras classes executarem as ações necessárias para o funcionamento do jogo.

Dentro do paradigma MonoGame a classe Game é executada ao iniciar o programa, primeiro rodando o método LoadContent, que carrega em memória todos os *assets* que serão utilizados, e logo em seguida o método Initialize que serve para iniciar todos os parâmetros necessários para o jogo incluindo o tamanho da tela, a cada frame os métodos Update e Draw também são executados, Update lidando com a lógica do jogo e Draw sendo utilizado para o desenho da tela

O pseudocódigo que ocorre ao iniciar o jogo é apresentado no algoritmo 1.

```
Carrega o conteúdo em memória
//Inicialização dos parâmetros de telas
```

Estabelece o tamanho de tela
Aplicas as mudanças
Criar o objeto câmera
Criar o objeto jogador
Estabelece o nível atual
Cria o objeto para Gerar números aleatórios
//Esse objetos será utilizado apenas para a criação de níveis, inimigos e
//outros objetos que precisariam de chamadas de RNG constantes teriam
//o seu próprio Objeto separado
Cria o mapa do primeiro nível
Cria e distribui inimigos no nível
Cria e distribui ouro pelo nível

Algoritmo 1 – Pseudocódigo para o início do jogo

O método é necessário para dar início e sempre vem antes do primeiro Update ou Draw.

O pseudocódigo do que ocorre a cada frame do jogo é apresentado no algoritmo 2:

//Update
Comanda o Player a fazer um Update
***Para cada** Inimigo*
    Faz um Update dele
***Para cada** Ouro*
    Faz um Update dele
***Se** o Jogador estiver morto*
    Reinicia o Jogo e suas variáveis
***Se** o Jogador estiver na Escada **e** com Ouro suficiente*
    Passa para o próximo nível
//Desenho
***Para cada** tile da grid*
***Se** o Jogador observou o Tile*
    ***Se** o Jogador está a menos 320 pontos de distância do tile*
        Desenha o Tile
    Senão
        Desenha o Tile, porém cinza
Executa Update no HUD com os novos valores
Desenha o jogador

Update da posição de Câmera

Para cada *Inimigo*

Se *o Jogador está a menos 320 pontos de distância do Inimigo*

Desenha o Inimigo

Para cada *Ouro*

Se *o Jogador está a menos 320 pontos de distância do Ouro* ***e*** *o Ouro* ***não*** *foi coletado*

Desenha o Ouro

Algoritmo 2 – Pseudocódigo para cada frame do jogo

A classe Game também mantem o número do nível atual, utilizando esse valor para criar mapas maiores, com mais ouro disponível, com mais inimigos, e com mais escadas disponíveis.

Classe Grid

A classe Grid cuida da criação do terreno do mapa, estabelecendo as Zonas e o espaço que elas ocupam e depois conectando as Zonas. A classe Grid também tem o terreno do mapa que é verificado por outras Classes, principalmente a classe Player, frequentemente

Para a geração de Zonas a Classe Grid o pseudocódigo apresentado no algoritmo 2 é utilizado.

Estabelece o tamanho vertical e horizontal de cada zona

Para Cada *zona a ser criada*

Seleciona se vai criar uma Zona TDML e TDCL

Se *é TDML*

Cria e gera uma Zona TDML

Senão

Cria e gera uma Zona TDCL

//Conecta as Zonas

Se *tem mais de um quarto*

Para cada *Zona*

Conecta a zonas adjacentes

//Cria Escadas

Para *(0 até Nível)*

Cria uma Escada

//Geração das paredes externas dos quartos e zonas

Para *cada tile do grid*

Se *é um tile de Vazio* ***e*** *possui um tile Chão adjacente*

Muda o Tile Vazio para Parade

Algoritmo 3 – Pseudocódigo para criação de grid

A geração das paredes externas ao quarto é apenas feita depois que todos as zonas e quartos foram preenchidos por ser uma solução simples e eficiente, é que feito antes do final da geração completa do espaço físico poderia gerar falhas no terreno.

Classe Zone

As classe Zone é criada pela classe Grid, ele ser como super Classe da ZoneTDML e ZoneTDCL. A zone contém métodos que lidam com a criação de quartos dentro do espaço da zona, a conexão de quartos por corredores e vãos.

Toda a Classe Zone tem uma variável constando o número de secções que elas têm, cada secção é um espaço reservado para um quarto ocupar. O tamanho dessas secções é calculado a partir do tamanho da Zona divido pelo número de secções.

Dentro a função da classe Zone também possui um método abstrato de geração dos quartos dentro da zona. A classes filhas, Zone TDML e ZoneTDCL, cuidam da geração de quartos dentro dos seus parâmetros.

O pseudocódigo da geração de uma Zona TDML é apresentado no algoritmo 4.

Estabelece o espaço horizontal e vertical de cada secção

Para Cada *secção*

Se *é a primeira secção na horizontal*

Quarto é posicionado no eixo horizontal

Se não

A aresta esquerda do quarto é igual a aresta direita do quarto a sua esquerda

Se *é a primeira secção na vertical*

Quarto é posicionado no eixo horizontal

Se não

A aresta superior do quarto é igual a aresta inferior do quarto acima dele

Define as outras arestas do quarto utilizando o tamanho esperado de cada secção como base

Criar o quarto usando as medidas estabelecidas

Se *é o primeiro quarto a ser criado na Zona*

Defina que o quarto está conectado

Constrói as paredes entre os quartos

Conecta os quartos

Algoritmo 4 – Pseudocódigo para geração da Zona TDML

Verifica se tem algum quarto que não foi conectado ao primeiro quarto ou um quarto já conectado a ele, conecta esses quartos *a todos os seus vizinhos*

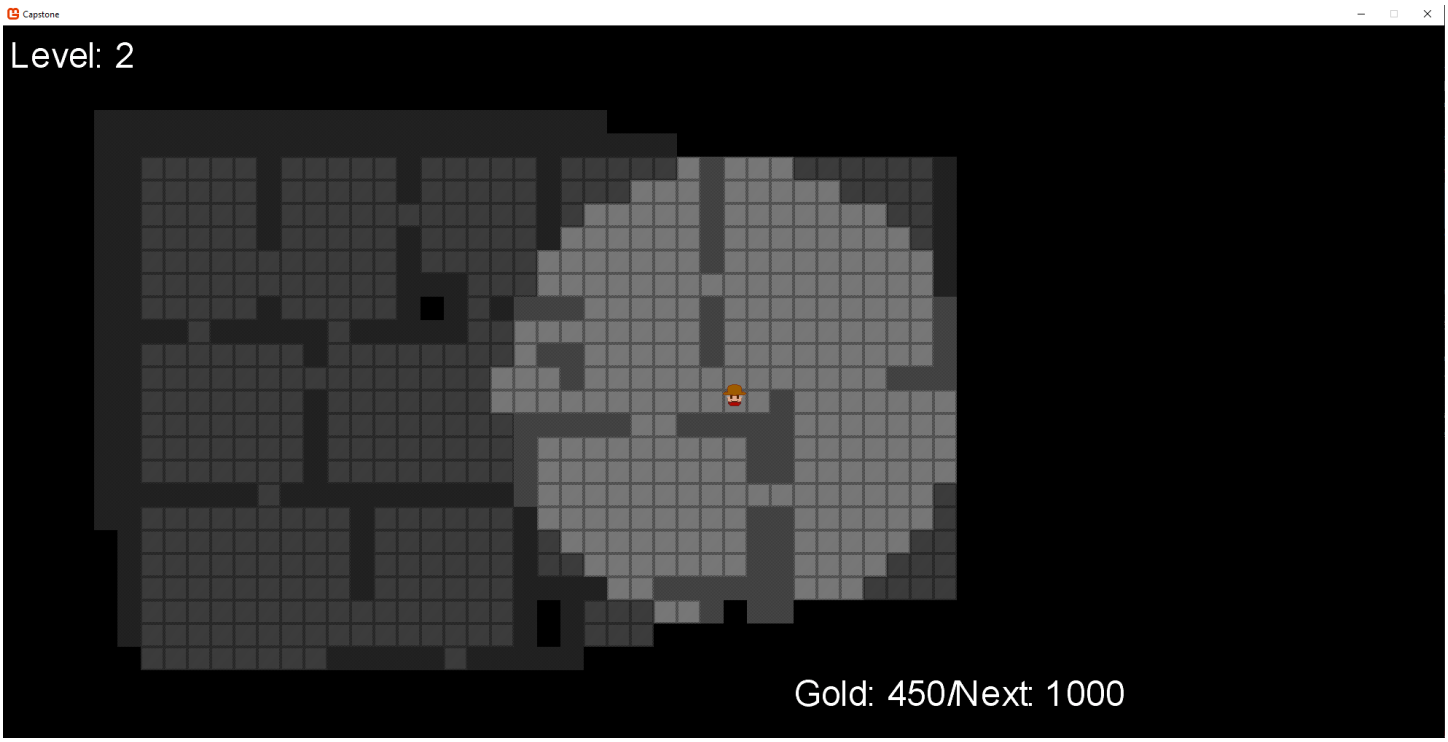

Figura 8 Zona TDML

Na Fig. 8 podemos ver como os quartos são compactos e separados apenas por um vão em suas paredes.

O pseudocódigo da geração de uma Zona TDCL é apresentado no algoritmo 5.

Estabelece of espaço horizontal e vertical de cada secção

Para Cada *secção*

Escolhe um o ponto esquerdo superior do quarto baseado no número do quarto e o tamanho da secção.

Aleatoriamente escolhe um ponto direito inferior baseado no primeiro ponto

Cria e constrói o quarto

Se *é o primeiro quarto a ser criado na Zona*

Define que o quarto está conectado

Conecta os quartos

Verifica se todos os quartos foram conectados e conecta aqueles que não foram.

Algoritmo 5 – Pseudocódigo para geração da Zona TDCL

Fig. 9 Zona TDCL

Como podemos ver na figura 9, em comparação com a figura 8, os quartos em TDCL são mais espalhados e conectados apenas por corredores, os limites dos quartos existem independentes das posições exatas de quartos anteriores

A classe Zone também é responsável por fazer a conexão entre dois quartos, ela pega um ponto dentro de cada um dos quartos e criar um corredor que conecta esses dois pontos.

Os corredores têm 3 métodos de criação, um que prioriza a coordenada X e depois a Y, um que o reverso, priorizando a coordenada Y e depois a X, e um onde a coordenada Y é priorizada, porém o corredor é maior e mais espaçado.

Para certificar que todos os quartos podem ser acessados a seguinte operação é feita, elaborando que foi mencionado no pseudocódigo.

O primeiro quarto gerado em qualquer zona é definido como o quarto de início, esse quarto de início é considerado conectado ao quarto inicial, todo quarto que está sendo conectado a um quarto que é considerado conectado ao inicial, também acaba sendo considerado conectado ao quarto inicial, pois significa que a navegação dele até o primeiro quarto é possível.

Assim conexão com um quarto considerado conectado “infecta” o quarto que está a conectar, se no final da conexão existe um quarto que não foi infectado, ele é conectado a um quarto adjacente que já foi verificado anteriormente.

Como a conexão e a checagem dessas conexões acontecem na mesma orientação sempre, é garantido que ao conectar com um quarto anterior, o quarto irá se tornar acessível ao jogador.

Um pseudocódigo para a conexão entre dois quartos é demonstrado pelo Algoritmo 6.

```
Se Quarto 1 ou Quarto 2 está conectado ao quarto inicial
    Ambos são conectados ao quarto inicial
Seleciona um ponto dentro do Quarto 1
Seleciona um ponto dentro do Quarto 2
Cria um caminho conectando os dois pontos
```

Algoritmo 6 – Pseudocódigo para criação de conexões

O pseudocódigo para fazer a checagem da conexão e conectar qualquer quarto ou bloco de quartos que estejam avulsos é apresentada pelo Algoritmo 7.

```
Para cada quarto da zone //Primeiro no eixo horizontal e depois vertical
    Se o quarto não está na posição X = 0
        Conecte ele a um quarto adjacente a sua esquerda
        //Verificado previamente
    Se não Se o quarto não está na posição Y = 0
        Conecte ele a um quarto adjacente superior
        //Verificado previamente
```

Algoritmo 7 – Pseudocódigo para verificação de conexão entre todos os quartos

Seguindo essa lógica, independente de quaisquer alterações causadas em como a Zonas fazem as conexões desses quartos, podemos verificar que dentro de uma Zona, um quarto pode ser caminhado até qualquer outro.

Classe Quarto

A classe quarto é relativamente simples, cada quarto é composto de dois pontos, um no canto superior esquerdo e um no canto inferior direito. Esses dois pontos são utilizados para criar o quarto, portanto os quartos apenas ocupam espaços retangulares.

Quartos também tem uma variável para identificar se corredores ou passagens os conectam com todos os outros quartos, porém como foi mencionado anteriormente, essa informação é utilizada pela zona para conectar os quartos.

Classe Player

A classe Player é uma das mais importante do jogo devido ao fato que é como podemos interagir com a *dungeon* gerada. O jogador pode navegar a *dungeon* utilizando a setas do teclado.

O jogo a cada frame prevê onde o movimento do jogador iria levá-lo e bloqueia o movimento ser ele iria colidir com alguma parede ou outro possível obstáculo o jogo tenta utilizar uma fração da velocidade atual para aproximar o jogador da parede sem colidir, ele tenta quatro reduções da velocidade correspondente a 100%, 75%, 50%, 25%, se em nenhuma destas o jogador encontrar um espaço que ele pode se mover para o movimento é cancelado.

Isso permite que o jogador se aproxime de uma parede mesmo que seu movimento total naquele frame normalmente iria impedi-lo de se aproximar.

A classe Joga também responsável por manter a Grid Visual, essa uma *grid* de valores bool equivalentes a grid do jogo, todas vezes que um tile aparece na visão do jogador ele é adicionado como no verdadeiro na Grid Visual. Ao jogo resetar ou ao mudar de nível o jogador recebe uma nova grid visual com o mesmo tamanho da nova grid gerada pelo jogo.

Como podemos ver na figura a seguir (Fig. 10), o Jogador tem uma área de visão, tudo que cai nessa área é adicionado a grid visual. Tudo os tiles que foram adicionados ao grid visual são exibidos, porém apenas os mais próximos o jogador é exibido na cor natural deles, todos os mais distantes são exibidos em uma cor cinza.

Fig. 10 Jogador e a grid visual.

Classe Gold

A classe Gold é uma classe simples, contendo a posição atual do Ouro, se ele já foi coletado, e o seu valor.

Também existem métodos para o ouro ser coletado e dar ao jogador o valor do ouro. A verificação dessa coleção, porém, não acontece na classe Gold. mas sim na classe Game.

Fig. 11 Jogador encontra uma barra de ouro.

Classe Monster

A classe Monster é responsável pelos inimigos, mantendo valores da posição, estado atual de ação, sua velocidade, e o nível do jogo atual. A classe Monster contém um fluxo de estados mais complexo para caracterizar o seu comportamento e dar mais dificuldade e inteligência ao inimigo.

O inimigo tem três estados que ele pode estar a qualquer momento.

A primeira é o estado “Idle” onde ele não estar a procurar ou perseguir o jogador, ele simplesmente ficar perambulando aleatoriamente e ficando ocioso por uns instantes.

O comportamento Idle é definido por duas variáveis principais, o “idleTimer” e “movementTimer”, quando ele está completamente parado o idleTimer conta os frames até ele voltar a mover, quando ele volta a mover, o inimigo move em uma direção aleatória por uma quantidade de frames definidas aleatoriamente no movementTimer, quando o movementTimer esgota, o idleTimer é resetado e o ciclo continua.

O estado Idle também interage com uma chamada “searchTimer”, essa variável serve apenas para caso o inimigo esteja muito distante do jogador por muito ele irá se direcionar para uma posição próxima do jogador no momento que o tempo de espera se esgotou, garantido que o jogador sempre estará sendo perseguido.

A classe Monster ao contrário do jogador, pode atravessar paredes com o seu movimento, isso devido dentro do contexto do jogo estarem sendo representados por fantasmas que não dão tanta importância a objetos físicos (além do jogador).

O pseudocódigo da máquina de estados e movimento dos inimigos é o seguinte

```
Caso estado atual é idle
    Decresce of searchTimer
    Se o searchTimer é 0
        Escolha uma posição perto do jogador como alvo
        Entra no estado de searching
        Reseta o searchTimer
    Se o Jogador está a menos de 200 pontos de distância
        Entre no estado de perseguição
        Reseta o searchTimer
    Se o idleTimer é 0
        Se o movementTimer é 0
            Para o inimigo de se mover
            Reseta o idleTimer
        Se não, Se o inimigo está a se mover
            Move o inimigo
            Decresce of movementTimer
        Se não
            Escolha uma direção para ser mover
            Escolha uma quantidade de tempo para mover
            Começa a mover o inimigo
    Se não
        Decresce o idleTimer
Caso o inimigo estiver no estado de perseguição
    Se o jogador estiver a mais 400 pontos de distância
        Estabelece a última posição que viu o jogador para investigar como alvo
        Reseta o searchTimer
        Muda o estado para busca.
    Se não //Que dizer que o jogador está perto
        Move o inimigo em direção ao jogador
Caso o inimigo estiver no estado de busca
```

__Se__ o jogador estiver a menos de 400 pontos de distância

Entra no estado de perseguição

Reseta o idleTimer

Reseta o searchTimer

__Se não__ //O jogador está longe

__Se__ o inimigo estiver próximo do alvo

Entra no estado idle

Reseta o idleTimer

Reseta o searchTimer

__Se não__

Move o inimigo em direção ao alvo

//Nota que o alvo pode ser a última posição do jogador

//Ou uma posição próxima do jogador

Confirma o movimento do inimigo

Verifica contato com o jogador.

__Se__ tem contato com o jogador

Mata o jogador

Algoritmo 8 – Pseudocódigo para o estado dos inimigos

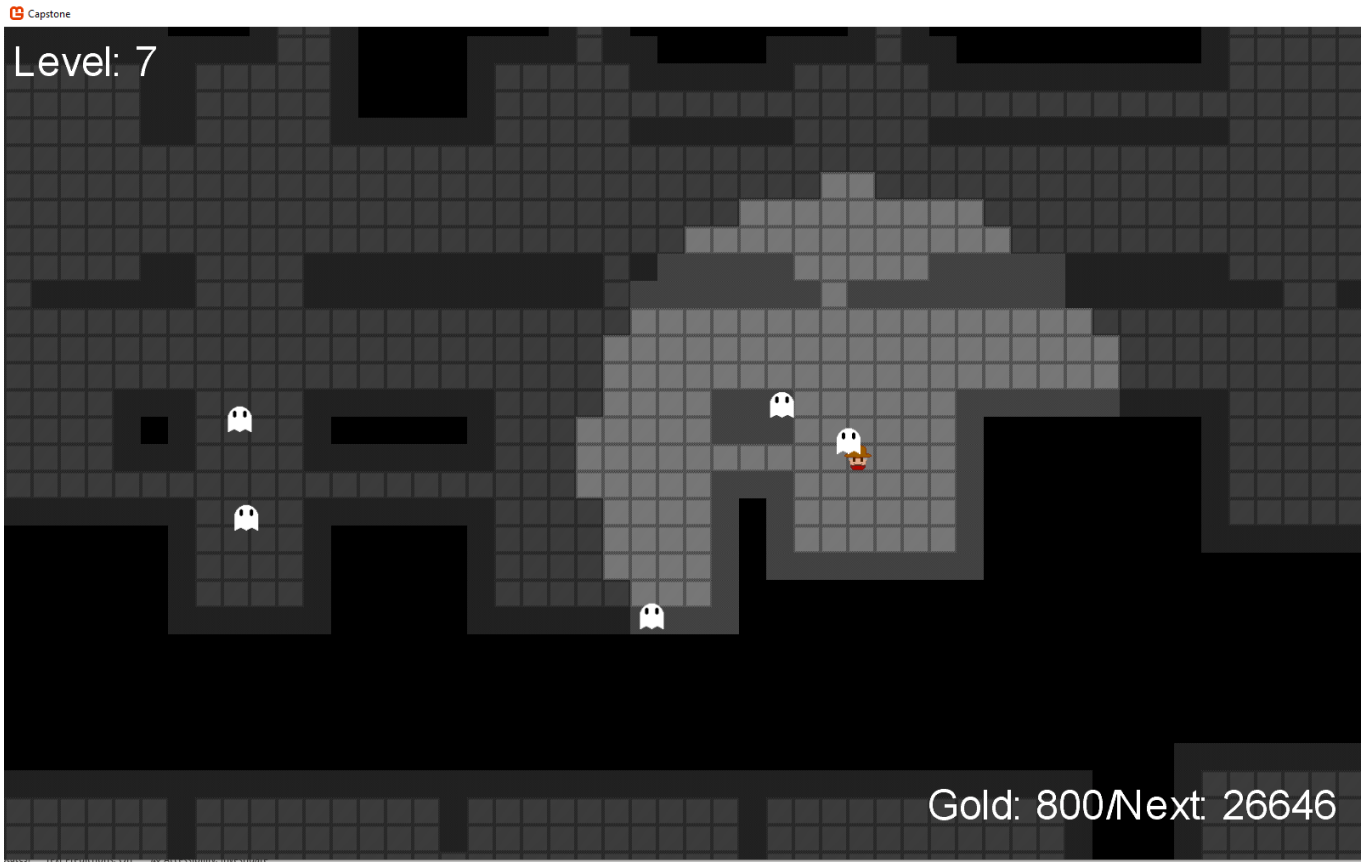

Fig. 12 Jogador sendo perseguido por inimigos.

Nessa figura (Fig. 12), apenas os inimigos mais próximos estão perseguindo o jogador, aqueles mais distantes (visíveis apenas por utilização do modo Debug) não sabem onde o jogador está, portanto estão apenas a perambular no seu estado *Idle*.

6. Implementação e Avaliação

Testes Práticos

Durante o desenvolvimento do projeto ficou evidente que a melhor forma de entender a geração de *dungeon* e reconhecer erros e defeitos na geração era por meios práticos, onde os erros observados seriam claros.

Portanto, a vasta maioria dos testes utilizados foram testes práticos e não automatizados, aonde a geração das zonas e quartos e a conexão entre eles eram examinados diretamente e quando erros eram observados na geração poderiam ser investigados com base da natureza do erro observado.

Durantes esses testes práticos limites sobre a observação da *dungeon* e a movimentação do jogador eram removidos usando um modo debug que permite passar por paredes, revelar o mapa inteiro, acelerar a velocidade do jogador e avançar o nível.

Os testes consistiam em criar pequenos mapas onde a funcionalidade dos métodos criados poderia ser testada e qualquer erros observados. Cada uma desses mapas teriam os objetos (quartos, corredores, zonas, inimigos) que estão sendo testados e bem visíveis. O comportamento desses era observado, e se algo erro acaba-se presente, o erro era investigado e removido antes de progredir para uma etapa futura do desenvolvimento.

Durante o processo diversos pequenos e grandes erros foram observados e concertados, esse processo foi repetido inúmeras vezes durante o desenvolvimento.

Comentário sobre a implementação

MonoGame como *Engine* para o desenvolvimento do Labirinto foi excelente para a implementação do projeto, sendo um ambiente simples de trabalhar, sem a necessidade de aprender a manusear um editor de cenas, como é feito em Unity, o fluxo e pipeline de desenvolvimento permitiu muito mais tempo ser dedicado ao desenvolvimento do projeto do que para aprender a utilizar o ambiente.

Para utilizar tudo que o ambiente do MonoGame oferecia foi apenas necessário conhecimento da linguagem C# e como a importação e carregamento de *assets* funcionava para o MonoGame.

O processo de desenvolvimento do projeto após estabelecer o MonoGame começou pela criação da classe Grid, dos quartos e do jogador, criando uma funcionalidade básica para explorar e testar a geração dos quartos e da grid. Logo após a geração dos quartos foi criado um método para conectar esses quartos formando o básico que seria usado no futuro para a criação das zonas.

Com os quartos e a capacidade de conectara-los completa, foi feita a implementação da Zona TDCL e a conexão entre zonas. TDCL foi escolhida como a primeira zona para ser desenvolvida devido ao fato de ser, com as ferramentas no momento, a Zona naquele momento do projeto parecer mais fácil de implementar e desenvolver o algoritmo e também mais simples de investigar erros, particularmente com a geração de corredores e quartos.

A próxima etapa no processor da implementação foi TDML, que foi mais complicado que a TDCL e tiveram inúmeros problemas devido a lógica de manter os quartos lado a lado sem ter problemas com um invadir o espaço do outro e durante a implementação foi descoberto um erro com a geração de corredores que apesar de estar presente no TDCL, era mais evidente nas zonas TDML e causava a zona a não ser explorável pelo jogador devido a maioria dos quartos não tendo saída.

De todas as etapas durante o desenvolvimento definitivamente a geração TDML foi a mais complicada e ofereceu mais desafios, incluindo pelo menos três vezes que foi necessário repensar como fazer a geração.

Com as zonas concluídas, a próxima parte do desenvolvimento foi a criação de inimigos para popular a *dungeon*, essas etapas foram razoavelmente sem problemas, apesar de ajustes aos valores de velocidade e quantidade de inimigos sendo feito até os últimos momentos do desenvolvimento.

O mais interessante sobre os inimigos foi a criação da uma Inteligência Artificial (IA) , onde o inimigo procura a última posição que ele tinha observado o jogador, e a IA *Idle* onde o inimigo perambula sem alvo. Ambas apresentaram desafios curtos, porém interessantes, os problemas que apareceram foram rapidamente e eficientemente contornados.

Um aspecto impressionante sobre o jogo, é a velocidade de geração de *dungeon* e o *update* dos inimigos, ambas, mesmo em condições extremas, operam sem atrasos ou travar, em termos práticos a geração de cada nível e a lógica de frame ocorre instantaneamente.

Comentários sobre Assets

Todos os *assets* artísticos utilizados para o Labirinto foram feitas pessoalmente utilizando o programa Krita[16], um programa grátis e *Open-Source* que eu já tinha uma familiaridade grande com.

Todos os assetes foram feitos dentro de um espaço quadrado de trinta e dois por trinta de dois pixels, e a parte gráfica e até mesmo mecânica do jogo foi feito em mente que a distância de um quadrado para o outro seria de trinta e dois pixels.

Arte foi feita com ume certa simplicidade para representar o terreno do mapa gerado, os inimigos e o ouro de forma bem visível. A paredes cinza-escuras fazem um contraste bom com o cinza-claro do chão, a aparência branca dos inimigos (que são fantasmas) é instantaneamente distinta do chão e parede e o amarelo do ouro não pode ser confundido com qualquer outra coisa.

As escadas também são inclusas tendo cores e padrões que com apenas um olhar informam que são diferentes do resto da dungeon.

7. Considerações Finais

Considerando o escopo e a intenção do projeto de desenvolver uma demonstração de geração procedimental, o projeto foi um sucesso. Existem claras áreas que o projeto poderia ser expandido e revisitado, porém o resultado atual, por mais simples que seja, ainda serve bem como uma exploração e demonstração de geração procedimental.

O processo de desenvolvimento foi um processo de aprendizagem com muitos desafios e problemas e apesar que ser ter sido possível alguns objetivos que pertenceriam a um escopo maior. O resultado é satisfatório.

Planos Futuros

Para o futuro do Labirinto, caso fosse dar sequência no desenvolvimento teriam pelos menos quatro tópicos de interesse para desenvolver e melhorar.

O primeiro tópico é a geração de dungeons mais complexas para a navegação, utilizando portas e chaves para conseguir acessar todos os quartos. Atualmente as dungeons são relativamente simples, passagens são sempre abertas ao jogador para passar, e maioria dos quartos são conectados pelo menos com os quartos adjacentes a eles.

Portanto seria de interesse adicionar bloqueios em formas de portas que poderiam ser desbloqueadas com chaves localizados no mapa ou em mapas

anteriores, ou contornados, sempre tendo em mente que o nível teria que ser possível de completar independente do estado que o jogador entrar nele, pois é necessário que a geração de dungeon não acabe por prender o jogador no nível.

Para isso seria necessário a construção de um algoritmo mais sofisticado para a geração, por exemplo, bloqueando a saídas de uma zona com portas trancadas, essas portas poderiam ser abertas com chaves dentro da zona, ou apenas uma das saídas da zone é fechada, dando uma vantagem a um jogador que acumulou chaves de um nível anterior.

Outra mudança que poderia ser realizada para dar mais variedade e complexidade ao layout da dungeon seria o posicionamento de quartos e o formato dos corredores também poderiam ser mais diferenciados, atualmente todos os quartos dentro das suas zonas são posicionados dentro blocos bem definidos, dando um aspecto menos variado a dungeon.

O sistema de zonas e quartos atual poderia comportar quartos posicionados menos rigidamente, porém poderia colocados em zonas que são variações das zonas TDML e TDCL, possivelmente sendo esses tipos diferentes desses tipos de zonas. Também para esses novos tipos de zonas possivelmente teriam que ser criados novos métodos para garantir que todos os quartos estão conectados, se fosse o caso que quartos poderiam não estar adjacentes uns ao outros.

O segundo tópico seria a criação de inimigos mais complexos, atualmente os inimigos praticamente ignoram a dungeon gerada, eles atravessam as paredes e ignoram quartos e itens.

Para o futuro seria interessante investigar uma IA que conseguiria perseguir o jogador com as mesmas limitações que o jogador tem, tendo que navegar pela dungeon respeitando as paredes. Esse inimigo com essa IA pertenceria a novas classe de inimigo que iriam coexistir com a classe “fantasma”, e daria maior diversidade nos desafios apresentados no jogo e seria um exercício interessante na criação de um algoritmo que fosse capaz de navegar a dungeon independentemente de como foi gerada e de perseguir o jogador.

Para essa IA seria necessário um algoritmo forte para detectar corredores e passagens e continuar a seguir o jogador quando ele passa de um quarto para outro, ou ampliar e modificar a geração de zonas, corredores e quartos, uma das opções seria a criação de nodes em cima do terreno criado, dando direções para a IA reconhecer quartos, corredores e outras passagens entre eles para navegar o mapa.

Também seria interessante adicionar comportamento para o inimigo “proteger” certos itens e quartos, por exemplo, teriam tipos de inimigos que

permaneceriam perto das escadas de saídas, sendo necessário para o jogador forçar o inimigo sair ou ser desabilitado para poder passar para o próximo nível. Outra situação poderia ser que inimigos ao estarem próximos de ouro ou algum outro item entram em ou estado onde eles orbitam esses itens, esperando pelo jogador se aproximar desse item.

O terceiro tópico é a adição de mais itens e mecânicas para ampliar as opções do jogador e habilidade sobreviver e reagir a inimigos.

Por exemplo dar ao jogador a ter uma “barra de vida”, onde o jogador poderia entrar em contato com o inimigo algumas vezes antes de “morrer” e o jogo ser resetado. Com uma barra de vida, também poderiam ser adicionados itens que poderiam ser encontrados e coletados no mapa para curar o jogador.

Por exemplo distribuir pelo mapa e permitir o jogador coletar munições para disparar nos inimigos, possivelmente imobilizando ou eliminando inimigos do nível.

O quarto tópico seria o aspecto visual da dungeon pois cada seção da dungeon é pouco distinta de outras seções devido ao fato que as paredes e chão tem a mesma cor. Em vários outros jogos é comum estabelecer biomas onde o assets gráficos são modificados para representar uma mudança na dungeon ou apenas para dar mais variedade ao espaço.

Apenas modificar a cor utilizada para a geração de zonas inteiras poderia ser um meio também de ajudar o jogador se localizar que seria mais bem apreciado em níveis mais altos onde o tamanho da dungeon e quantidade de zonas cresce de forma significativa.

A ideia de biomas foi algo que surgiu durante o desenvolvimento e apesar de ter tido algo tempo dedicado, foi abandonado devido a complicações e necessidade de colocar mais foco e partes mais essenciais do projeto, porém é com certeza algo que seria de interesse para retornar caso o projeto fosse continuado.

Biomas também poderiam ser usado com outros planos futuros, por exemplo, inimigos que só aparecem em certos biomas ou que tem sua movimentação limitada pelo bioma que habitam.

Outras adições que seriam interessantes seria objetos meramente decorativos que não adicionariam nada as mecânicas, mas poderiam ser localizados dentro das zonas para decorar elas, usando esses juntos com biomas, esses objetos decorativos poderiam ser espalhados dependendo de biomas, apenas sendo presentes em alguns e ou sendo mais ou menos frequentes dependendo do bioma.

8. Referências

[1] Viana, Santos: Procedural Dungeon Generation: A Survey, 2021.

[2] Sampaio, Baffa, Feijó, Lana: A fast approach for automatic generation of populated maps with seed and difficulty control, 2017.

[3] Short, Adams: Procedural Generation in Game Design, 2017.

[4] Whitehead: Spatial Layout of Procedural Dungeons Using Linear Constraints and SMT Solvers, 2020.

[5] Gellel, Sweetser: A Hybrid Approach to Procedural Generation of Roguelike Video Game Levels, 2020.

[6] Gutierrez, Schrum: Generative Adversarial Network Rooms in Generative Graph Grammar Dungeons for The Legend of Zelda, 2020.

[7] Var der Linden, Lopes, Bidarra: Procedural Generation of Dungeons, 2014.

[8] Kreitzer, Ashlock, Pereira: Automatic Generation of Diverse Cavern Maps with Morphing Cellular Automata, 2019.

[9] Sheffield, Shah: Dungeon Digger: Apprenticeship Learning for Procedural Dungeon Building Agents, 2018.

[10] Tutenel, Bidarra, Smelik, Kraker: The Role of Semantics in Games, 2008.

[11] Monogame, Disponível em: <https://docs.monogame.net/index.html> (Acesso em maio, 2023).

[12] RogueSharp, Disponível em: https://roguesharp.wordpress.com (Acesso em Junho, 2023)

[13] Unity Engine, Disponível em: https://unity.com/products/unity-engine (Acesso em Novembro, 2023)

[14] Discover Game Development with C# Programming and Monogame, Disponível em: https://www.udemy.com/course/monogame/ (Acesso em Novembro, 2023)

[15] Comora https://github.com/dotnet-ad/Comora (Acesso em Novembro, 2023)

[16] Krita https://krita.org/en/ (Acesso em Novembro, 2023)

[17] Brycer, Joshua: Game Design Deep Dive – Roguelikes, 2021