UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

ALGORITMOS ANYTIME EM VIDEOJOGOS

João Pedro Silva Fernandes Batista

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Interação e Conhecimento

Dissertação orientada pelo
Prof. Doutor Luis Manuel Ferreira Fernandes Moniz
e pelo Prof. Doutor Paulo Jorge Cunha Vaz Dias Urbano

2017

Agradecimentos

Aos meus Pais e ao meu Irmão pelo apoio que me proporcionaram ao longo deste ano, garantindo, que me mantinha no caminho certo e que dispunha de todos os recursos necessários à realização deste trabalho, acreditando sempre nas minhas capacidades e incentivando-me diariamente a melhorar.

Aos meus colegas e amigos, Júlio Machado, Fábio Constantino, Diogo Mendonça, André Loureiro, Pedro Gomes, João Deus, Denis Cunha e Pedro Cabo, e às meninas Marta Silva e Mylene Pedrosa pela paciência e apoio que me foram prestados ao longo deste ano.

Quero ainda agradecer ao Fernando Fernandes, Fernando Alves, André Rosa e José Simões pelo apoio que me deram ao longo deste ano, com um especial obrigado ao senhor Frederico Ferreira, pela ajuda proporcionada durante o desenvolvimento da componente em C# do projeto.

Aos orientadores do projeto, Prof. Luís Moniz e Prof. Paulo Urbano pela oportunidade de trabalhar sob o seu olhar atento, querendo ainda agradecer à Prof. Ana Paula Cláudio pela disponibilidade demonstrada ao longo do desenvolvimento do trabalho realizado.

Aos sonhadores, que trabalham incessantemente para atingir os objetivos que nas suas vivencias aparentam ser longínquos.

Resumo

Desde a criação do primeiro videojogo há aproximadamente 60 anos, que estes têm servido como método de implementação de conceitos e algoritmos inovadores. A necessidade que as empresas de publicação e desenvolvimento de videojogos sentem em manter um jogador interessado nos jogos que desenvolvem é mais do que visível. Torna-se por isso necessária a implementação de fatores de aleatoriedade, que afetam os inúmeros elementos que compõem um jogo através da inserção de variações no jogo original. Usa-se, para o efeito, a geração de terrenos como uma forma de permitir a criação de um elemento que dê aos jogadores novas experiências de jogo de forma constante.

Os algoritmos Anytime permitem obter uma solução válida em qualquer ponto da execução, conseguindo melhorar os resultados obtidos à medida que o tempo de computação passa. Isto confere a estes algoritmos um nivel de flexibilidade único, especialmente quando são aplicados à geração de terrenos, pois conseguem melhorar de acordo com o tempo de execução.

Um Sistema Baseado em Regras representa conhecimento por regras semelhantes à linguagem natural. Nestes Sistemas o conhecimento é organizado sob a forma de uma série de regras que influenciam o comportamento do programa provocando, subsequentemente, alterações na execução do código. A Programação dinâmica é regularmente aplicada na área da Inteligência Artificial de modo a permitir que soluções futuras a um problema possam ser inferidas a partir de soluções previamente obtidas, conseguindo resultados melhores à medida que o tempo de execução vai aumentando.

Partindo deste pressuposto, este projeto apresenta um algoritmo que permite a geração de terreno em "Anytime", procurando manter-se a coerência e eficiência necessárias ao funcionamento do jogo, aplicando conceitos de programação dinâmica para garantir um aumento na "replayability" do jogo, visando facilitar a tarefa de gerar um mapa de acordo com um dado conjunto de regras.

Palavras-chave: Anytime, Geração de Mapas Procedimental, Unity 3D, Videojogos, Sistema Baseado em Regras

Abstract

Since the creation of the first videogame 60 years ago, these have served as a way to implement innovative concepts and algorithms, applicable to multiple areas other than entertainment. Videogame publishing and development companies feel a necessity towards keeping players interested in a game no matter the ammount of time they have spent. As such they now turn to the implementation of randomness as a factor, affecting the numerous elements that compose a videogame thus causing the original game to have inumerous possible variations. To this end, terrain generation is used as a means to develop an element that can constantly provide players with new experiences.

Anytime algorithms always return a valid solution independently of the current point of the algorithm's execution, being able to improve on previously obtained results as computation time increases. This grants these algorithms an unique level of flexibility, especially when applied to terrain generation, due to their ability to constantly improve.

A Rule Based System represents knowledge through rules similar to natural language. In these systems knowledge is depicted as a series of rules which influence the program's behaviour causing, subsequently, changes in the code's execution which afect the final result in a unique manner. Dynamic Programming is usually applied in Artificial Intelligence in order to allow the inference of new solutions using previously obtained results as a base, thus achieving better solutions as time goes on.

This project presents an algorithm that makes use of "Anytime" in order to generate terrain, while trying to maintain the coherence and efficiency required for a game to run, granting an increase to replayability through the implementation of dynamic programming elements whilst seeking to facilitate the task of generating a map according to a specific set of rules.

Keywords: Anytime, Procedural Map Generation, Unity 3d, Videogames, Rule Based Sistem

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Cada vez mais os videojogos estão presentes nas nossas vidas. À medida que foram evoluindo, o conceito de serem apenas para crianças foi-se dispersando, sendo hoje em dia raras as famílias que não lhes acedem independentemente das idades daqueles que as constituem, vulgarizando-se a sua utilização entre as gerações mais novas.

Com a introdução do primeiro videojogo há 60 anos [14], os jogos instalaram-se de tal forma no nosso quotidiano, que hoje em dia, é difícil pensarmos no nosso futuro sem eles e sem a realidade virtual que lhes está associada.

Os videojogos são, na atualidade, capazes de educar, de entreter, de ensinar e até mesmo providenciar emprego aos interessados na área. Mas acima de tudo, os videojogos impulsionam a evolução da ciência da computação tanto a nível do hardware como software, devido à necessidade de ultrapassar as limitações nos recursos disponíveis. A necessidade de estar na vanguarda da tecnologia, de fazer algo diferente e melhorar o que já existe, transformou a criação de videojogos numa área bastante competitiva. O desenvolvimento de um videojogo deve apresentar características que o transformem em algo que os consumidores sintam necessidade de ter. Por exemplo, um videojogo recém-desenvolvido deve, possuir: mecânicas interessantes; bons gráficos; uma IA (ou Inteligência Artificial) devidamente desenvolvida, afetando a forma como o software na área evolui pois é imperativo ter algo que um rival não tenha. Neste âmbito, a IA desempenha um papel fundamental no que toca a garantir a imersão de um jogador no jogo e facilita, por outro lado, o trabalho da equipa de desenvolvimento através da automatização de muitos dos processos usados. É neste âmbito que se destacam algoritmos que dão ao programador a capacidade de criar facilmente realidades virtuais que permitem captar a atenção do jogador. Uma forma de garantir que o jogador permanece interessado no jogo passa por fazer com que este esteja em constante mudança, podendo isto ser facilmente conseguido através da implementação de um gerador de terreno automatizado que gere novos terrenos de acordo com um valor (ou "seed") aleatório criado no inicio da execução do algoritmo.

A geração automatizada de terrenos em videojogos é algo que tem vindo a surgir com cada vez mais força nos anos que correm. Hoje em dia os jogadores requerem alguma coisa que mantenha o interesse no jogo [26]. A geração automatizada de terrenos ou mapas de jogo pode ajudar neste aspeto, graças à sua capacidade para criar um mapa diferente (por muito ligeiras que essas diferenças sejam) de cada vez que o algoritmo é executado, resultando assim numa experiência diferente de cada vez que alguém joga um jogo que faça uso de um destes algoritmos.

Visando gerar um terreno infinito há que causar variações durante o processo de geração, podendo ser usado para o efeito um Algoritmo Anytime. Os resultados destes algoritmos melhoram gradualmente à medida que o tempo de computação aumenta [35]. Este algoritmo funciona à base de um trade-off, uma vez que a qualidade da solução é proporcional ao tempo gasto pelo algoritmo. A grande particularidade de um algoritmo Anytime é a capacidade de devolver sempre, em qualquer ponto do tempo, uma solução válida (independentemente da qualidade da solução), significando que, através da sua aplicação, podemos ter inúmeros outputs diferentes para o mesmo input, podendo todos estes outputs serem considerados viáveis.

De modo a permitir um certo nível de flexibilidade na geração do terreno pode recorrer-se a Sistemas Baseados em Regras (ou SBR), que são dos sistemas mais antigos utilizados na área da Inteligência Artificial [10]. Neste tipo de algoritmos, o conhecimento é organizado sob a forma de uma série de regras que influenciam o comportamento do programa, provocando subsequentemente, alterações na execução do código. As regras, no entanto são apenas um dos elementos que constituem um SBR, sendo cada SBR normalmente constituído por um Motor de Inferência e uma Interface através da qual o utilizador pode interagir com o sistema.

Apesar de tudo isto e da idade que os SBRs já têm, noticias da sua aplicação na área dos videojogos é incomum e potenciais tentativas de fundir um SBR com elementos Anytime, aparentam, até a data, ser nulas. O Minecraft [7] é um exemplo de um jogo possuidor de um sistema de geração de terreno que é composto pela integração de um SBR num ambiente Anytime. Este sistema gera terreno apenas quando o jogador se aproxima de uma área vazia, construindo sempre uma solução válida usando um conjunto de regras predefinido para determinar qual o tipo de terreno a criar. Faltam-lhe, no entanto, muitos aspetos que compõem um algoritmo Anytime, como por exemplo, a capacidade de aprender a gerar uma solução melhor à medida que o tempo de execução aumenta.

Pretende-se assim criar um gerador de terreno que consiga fundir elementos Anytime com um SBR sob a forma de um gerador de terreno automatizado.

1.2 Objetivos

O objetivo principal deste projeto é construir uma aplicação que faça uso de algoritmo de decisão que funcione em Anytime (ou seja, que consiga formular sempre uma decisão viável independentemente do momento em que lhe seja pedida uma solução) utilizando um Sistema Baseado em Regras para armazenar o conhecimento adquirido. Pretende utilizar-se como exemplo um ambiente semelhante ao do Minecraft, sendo que neste caso, o algoritmo terá que decidir quais os blocos que serão colocados no plano de jogo de acordo com os dados que caracterizam o ambiente e o ponto em que estes deverão ser postos.

O Algoritmo final terá como objetivo cumprir os seguintes requisitos:

- Fazer uso de um conjunto de regras simples para tomar decisões iniciais;
- Aprender a lidar com situações novas de acordo com o conhecimento que o algoritmo já tem;
- Garantir consistência no que toca às decisões tomadas;
- Ser o mais generalizado possível, ou permitir a sua reimplementação de forma facilitada em novos casos;
- Evitar ter um impacto muito grande em termos do desempenho da aplicação;

Pretende-se também que o exemplo utilizado possua as seguintes características:

- Gerar terreno de forma procedimental;
- Seja extensível por outros utilizadores;
- Seja simples de implementar independentemente do ambiente em que se encontra;

Foi desenvolvida uma aplicação de exemplo em Unity pois esta plataforma é de fácil utilização e tem uma vasta quantidade de recursos "Open Source" disponíveis.

O objetivo inicial do projeto era construir um algoritmo fazendo uso de uma ponte entre o C# (usado pelo Unity) e Prolog (uma linguagem usada especificamente para programação lógica e matemática). Devido a fatores que serão mencionados posteriormente, tal não foi possível, pelo que o algoritmo de decisão final foi construido apenas em C#, devendo este ser aplicável num ambiente com uma versão da framework .Net igual ou superior a 3.5.

1.3 Contribuições

Com este estudo, apresenta-se um algoritmo que possa servir como uma introdução para a criação de potenciais algoritmos Anytime que funcionem usando elementos pertencentes a Sistemas Baseados em Regras. Atualmente, os recursos académicos encontrados que fazem referência ao uso do Anytime em jogos não são em grande número e apesar da utilização relativamente frequente de SBRs na criação de elementos de Inteligência Artificial especializados, ainda não se verificou a implementação de um algoritmo híbrido que conjugue as características existentes em cada um dos sistemas acima mencionados.

O suporte e implementação deste algoritmo de decisão sobre um gerador procedimental de terreno serve também para garantir que este algoritmo pode ser implementado em jogos que procurem aumentar a sua "replayability" através da implementação de elementos aleatórios, demonstrando assim que este pode ser implementado para resolver situações em que a potencial longevidade de um jogo está em causa.

Salienta-se, que apesar da informação de teor cientifico ser escassa, uma implementação de Anytime a este nível pode existir já num âmbito comercial. Infelizmente, o espaço que trata de IA neste plano é restrito, sendo a informação facultada para investigação reduzida, ou, como ocorre neste caso, nula.

1.4 Estrutura do documento

Este documento está organizado da seguinte forma:

- Capítulo 2 – Apresentam-se investigações relevantes aos campos de investigação que se encontram diretamente relacionados com o tema em estudo. Este capítulo encontra-se dividido em três subcapítulos, para o efeito, no primeiro apresenta-se uma introdução breve a Algoritmos Anytime usando como exemplo alguns estudos feitos na área; no segundo abordam-se os SBRs (também conhecidos como Sistemas Baseados em Regras ou "Expert Systems") providenciando informação sobre o que é que compõe exatamente um destes sistemas e de que forma é que estes podem ser aplicados na área dos videojogos; e, finalmente, o terceiro dedica-se à geração de terreno automatizada em videojogos e à informação relativa a alguns métodos e técnicas usados para este efeito.

- Capítulo 3 - Descreve-se o problema, a metodologia de investigação usada, explicitando o delineamento do estudo, os materiais usados e o planeamento do trabalho.

- Capítulo 4 - Apresenta-se o trabalho realizado em detalhe, onde se descreve de forma pormenorizada cada uma das componentes da implementação e se salientam os eventos de particular importância entre cada fase do desenvolvimento.

- Capítulo 5 - Procede-se à demonstração dos resultados finais obtidos.

- Capítulo 6 - Analisam-se e explicam-se os resultados obtidos no decorrer do estudo.
- Capítulo 7 - Apresentam-se as conclusões obtidas no final do estudo, referindo aspetos a melhorar em futuras ocasiões tendo como base os resultados obtidos neste trabalho.

1.5 Sumário

Este projeto implementa um algoritmo de aprendizagem automatizada em jogos fazendo uso de elementos Anytime e tendo como base características que são tipicamente usadas para descrever e implementar Sistemas de Regras. O algoritmo resultante tem como objetivo ser versátil o suficiente para poder ser facilmente implementado em situa-ções que fujam largamente ao exemplo dado, sendo apenas necessárias, para o efeito, ligeiras alterações no código. Pretende-se assim que o algoritmo seja eficiente o suficiente para não interferir com a execução de outros algoritmos que componham o resto da aplicação.

Desta forma, este estudo tem como objetivo principal a abertura de novos horizontes tanto na área da IA em Anytime como na da IA baseada em regras.

Capítulo 2

Trabalho relacionado

2.1 Algoritmos Anytime

De todos os algoritmos usados para a criação de IA, em jogos, os algoritmos que lidam com Anytime são, muito provavelmente, aqueles menos explorados a nível académico. No entanto, apesar da informação relativa a estes algoritmos ser um tanto escassa, existem algumas fontes que procuram delinear quais os elementos necessários à construção de um algoritmo que corra em Anytime, tentando inclusive, definir formas que permitam a sua manutenção e controlo. De acordo com as fontes consultadas, um algoritmo Anytime pode ser definido como um algoritmo que gera uma solução cuja qualidade é proporcional ao número de recursos consumidos [35], sendo a avaliação da qualidade de um algoritmo Anytime feita através da comparação entre a qualidade da solução face aos recursos consumidos pelo algoritmo[36].

Assim, um algoritmo Anytime deve ser desenvolvido de acordo com as seguintes métricas:

1. A *Certeza* a partir da qual se pode afirmar que uma solução gerada pelo algoritmo está correta;
2. A *Precisão* que este possui ao gerar uma resposta ao problema exposto;
3. A *Especificidade* (ou nível de detalhe) do resultado obtido;

Existe ainda um conjunto de características que devem ser tidas em conta na definição de métricas que visem avaliar a qualidade de um destes algoritmos:

1. A *Qualidade Mensurável*, que reflete a distância entre os resultados aproximados e corretos;
2. A *Qualidade Reconhecível*, apenas determinável ao correr o algoritmo, estabelece uma comparação entre a qualidade da resposta obtida face a respostas diferentes;

3. A *Monotonicidade*, ou seja, o resultado obtido melhora de acordo com o tempo e a qualidade dos dados fornecidos;

4. A *Consistência* dos resultados obtidos pelo algoritmo;

5. A *Capacidade de melhorar*, largamente, soluções iniciais no principio da execução do programa;

6. A *Interruptibilidade* do algoritmo, ou seja, a capacidade de ser interrompido a qualquer ponto da execução;

7. A *Capacidade de recuperação* de interrupções, mantendo a integridade dos dados;

A formulação de um algoritmo Anytime não deve perder de vista este conjunto de factores, tentando sempre manter o máximo de funcionalidade possível.

De forma a facilitar a manutenção e monitorização destes algoritmos, foi apresentada uma framework [36] que pretende controlar o comportamento do algoritmo sobre o qual corre. Esta framework analisa o funcionamento base do algoritmo de acordo com as características que o definem, obtendo estimativas de valores através da avaliação das características previamente mencionadas, bem como a partir de elementos que mudam durante a execução (como por exemplo estado do ambiente sobre o qual o algoritmo está a ser executado) lidando deste modo com o elevado número de fatores que influenciam o resultado final. Para além disso, há ainda que garantir que o algoritmo pode ser executado em "runtime" sem que a sua eficiência seja afetada, existindo por isso a necessidade de tratar de todos os problemas relativos à alocação do tempo de computação recorrendo para o efeito a uma sequência de previsões e decisões por forma a evitar a necessidade de ficar à esperar de soluções.

Um exemplo recente de um algoritmo Anytime refere-se à sua implementação num caso da vida real, sob a forma de um jogo de Gato e Rato [29]. O truque do jogo consiste na presença de um limite de tempo que define quando o jogo termina, sendo as soluções possíveis são restringidas de acordo com o tempo máximo disponível.

O principal problema tratado por esta aplicação está relacionado com o "tracking" baseado na visibilidade dos elementos presentes no mapa. De acordo com a teoria de jogos, este é definido como um jogo de perseguição/evasão com base na visibilidade de cada jogador, em que as jogadas são feitas em simultâneo.

Foram utilizados dois algoritmos diferentes (MinMax com profundidade iterativa e a Pesquisa em Árvore usando Monte-Carlo) no desenvolvimento deste sistema, tendo sido realizados 100 testes com uma duração de 100 passos cada. Foi também utilizada aleatoriedade de modo a que o jogador que foge esteja visível pelo menos a um dos perseguidores, estando no entanto, longe o suficiente para garantir que estes não o apanham imediatamente.

Os resultados finais desta experiência demonstraram que embora ambos os algoritmos sejam opções viáveis no que respeita à construção de um algoritmo Anytime, o MinMax demonstra ser uma opção melhor do que o Monte-Carlo quando está em causa o desenvolvimento de um algoritmo que vise criar um elemento que tem como objetivo comportar-se como um fugitivo.

2.2 Sistemas Baseados em Regras

De acordo com a Enciclopédia da Inteligência Artificial [31], um SBR (ou Sistema Baseado em Regras) é um tipo de Sistema Baseado em Conhecimento em que o conhecimento é representado por expressões, como por exemplo:

Algoritmo 1 Exemplo de uma Regra

if A
and B
then C

Estas regras podem ser escritas de forma muito semelhante a linguagem natural. Com base nestas regras são derivados factos que são inferidos através da interação entre as regras já definidas e outros factos pré-existentes. Um SBR permite assim a implementação de Inteligência Artificial que consegue tomar decisões de acordo com um conjunto de regras internamente definidas, comparando factos e executando regras que definem de que forma é que a IA se deve comportar. Interligando regras e factos é possível ainda estabelecer uma rede de inferência, que permite obter novas regras e factos que podem ser usados futuramente na análise de dados pelo sistema. Artigos de teor académico que mencionem a aplicação deste tipo de sistemas a videojogos são, no entanto, difíceis de encontrar, havendo a necessidade de ampliar a pesquisa relativa a este tema.

Nesta área, a existência de informação de particular interesse relaciona-se com a criação de um Sistema Baseado em Regras pretendendo que este sirva de suporte a um Sistema em de Decisão previamente existente [27] usado uma operação petrolífera. Neste caso, o sistema de regras foi implementado de modo a que este consiga atingir objetivos pré-definidos, algo que deve ocorrer através do estabelecimento de relações entre os casos presentes no sistema primário e as regras existentes no secundário, sendo estas relações obtidas através da tradução de casos base para regras, visando manter sob controlo possíveis falhas na representação de casos na base de casos.

O autor do projeto não conseguiu alcançar o objetivo pretendido, não tendo sido possível implementar um processo de "Case Matching" funcional. Este insucesso foi o resultado direto de falha no acesso aos casos da Base de Casos presentes na Base de Conhecimento sob a forma de uma rede semântica.

Encontra-se aqui também uma proposta de um método teórico de design de ferramentas de debug que servem para verificar e validar os dados presentes nas Bases de Conhecimento

utilizadas em SBRs, detetando potenciais anomalias que possam vir a afetar a consistência das mesmas.

A proposta de Petter Folgelqvist [28] descreve a forma como uma ferramenta automatizada para verificação de regras pode ser desenhada e desenvolvida de modo a que uma aplicação prática da mesma verifique e analise uma Base de Regras tendo em conta um conjunto de fatores predeterminados, que se encontram divididos em dois grandes grupos:

1. *Regras Redundantes*, que tornam o código ineficiente, causando, em certas ocasiões, alterações no comportamento;

2. *Regras "Unreachable"*, que nunca são executadas e que causam assim potenciais falhas no sistema;

Folgelqvist afirma, no entanto, que apesar deste método de design permitir a análise destes fatores, os erros provenientes de elementos responsáveis pela ordem da execução do código correspondente às regras devem ser analisados cuidadosamente pelo programador da aplicação de verificação e validação automática, requerendo assim um conhecimento profundo da implementação do sistema. Graças a este sistema de design inovador, tornou-se possível encontrar novos processos de decisão que podem ser utilizados no desenvolvimento de elementos de verificação e validação de Sistemas Baseados em Regras, algo imperativo para garantir a sua eficiência e bom funcionamento.

2.3 Geração Procedimental de Terreno

Pretendendo dar-se suporte a um sistema que esteja em evolução constante, é natural que se fale sobre Geração Procedimental. Sendo imperativo manter um jogador interessado num videojogo o máximo de tempo possível torna-se importante recorrer à geração procedimental de conteúdo aleatório e a geração de terreno cai sobre a categoria de elementos que pode variar com cada nova geração. A aplicação apresentada neste projeto deve providenciar um mundo que se possa expandir infinitamente, recorrendo-se para o efeito, a algoritmos de geração que permitam que o terreno seja gerado de acordo com a posição do jogador.

Artigos que descrevam este tema de forma detalhada são escassos, Gillian Smith dedica, no entanto, uma secção a este tema na segunda publicação do livro Ai Game Pro [33], não se focando exclusivamente na geração procedimental aplicada a terrenos, mas discutindo este tema ao nível de qualquer conteúdo que possa vir a ser gerado automaticamente em videojogos. Smith fala sobre diferentes métodos de desenvolver algoritmos que sirvam o propósito de gerar conteúdo de forma automática. São eles o *Baseado em Simulação*, construido com base num terreno básico inicial usando modelos (como por exemplo o de erosão e chuva) para gerar o restante terreno; o *Construtivo*, que usa blocos base para

gerar o ambiente de jogo, sendo estes blocos usados com regularidade na geração de níveis para jogos "Rogue-like", estando a IA "presente" nas escolhas feitas pelo algoritmo; o *Gramatical*, que usa uma gramática própria que estabelece quais os tipos possíveis de objeto que podem ocupar um espaço definido através do estabelecimento de determinadas regras, sendo estas posteriormente passadas por um interpretador que usa essas mesmas regras para gerar conteúdo; o de *Otimização*, que usa um processo de pesquisa para tentar obter uma combinação ótima de componentes de acordo com uma função de avaliação, podendo potencialmente recorrer a um elemento humano, tipicamente um designer, para ajudar a definir o conteúdo preferido de entre os presentes nos elementos disponíveis; o *Orientado a Constantes*, um método onde se restringe o design do algoritmo de modo a assegurar que ao ser executado este consegue todas as soluções potenciais que vão de acordo com as restrições impostas.

A representação do conhecimento usado pode também variar, estando os métodos para a representação de conhecimento divididos em quatro grandes categorias. Esta representação pode ser tratada criando "Experiential Chunks" que capturem uma quantidade de dados suficientemente grande para permitir a sua interpretação fora do contexto do jogo. Em vez de "Experiential Chunks" podem ser criados "Templates" que são uma forma mais generalizada destes "Chunks" onde a equipa de design do jogo tem controlo sobre o conteúdo presente no elemento criado, deixando no entanto, algum espaço que a IA pode preencher automaticamente de modo a criar variações no resultado da geração. Podem ainda ser criados "Components" que são fundamentalmente padrões desenhados por um elemento humano que contêm uma quantidade de dados limitada não podendo ser interpretados por si só, algo vantajoso quando não se quer que um jogador se aperceba da existência de padrões no processo de geração. Finalmente, a representação do conhecimento pode ser feita criando "Subcomponents" que são efetivamente a representação que menor quantidade de dados possui, devendo estes ser tratados como pequenos blocos usados em conjunto para gerar e construir novos elementos de jogo. Apesar da flexibilidade deste último método de representação de conhecimento, este é raramente usado por ser dificil ao gerador perceber como combinar cada "Subcomponent" sem a ajuda de alguma informação que explicite como o fazer. Um gerador procedimental deve assim, de acordo com Smith, recorrer a uma mistura de cada um destes elementos, pois estes permitem a criação de sistemas cada vez mais complexos e sofisticados, adequados às necessidades dos jogadores de hoje em dia.

A geração de terreno na criação de conteúdo dinâmico em jogos é discutida por Noor Shaker, Julian Togelius, e Mark J. Nelson no quarto capítulo do livro Procedural Content Generation in Games [34]. O grupo analisa múltiplos métodos de geração de terreno, começando por "Heightmaps", um método que recorre à divisão do mapa gerado em células e à atribuição de valores correspondentes à altura a cada célula. Usando este método como base, a geração de um mapa chega a ser simples ao ponto de ser apenas necessário atribuir um valor aleatório a cada célula, resultando a aplicação deste método,

no entanto, em mapas que não parecem naturais, ou seja, que não apresentam o mesmo tipo de relevo continuo que se encontraria na natureza. Visando mitigar o problema que advém da utilização de aleatoriedade na geração do relevo de terrenos é possível recorrer a múltiplos métodos de interpolação (cada um deles com o seu próprio "trade-off" entre a qualidade dos resultados obtidos e o tempo de computação necessário) aplicados aos valores aleatórios usados. Uma alternativa ao uso de "Heightmaps" são os chamados "Fractal Methods" ou Métodos de Fractais, como por exemplo o "Diamond-square", onde se encontram incluídas implementações de geração de terreno recorrendo a "Noise" ou Ruído. Estes métodos geram valores de altura de acordo com um conjunto de múltiplas escalas diferentes, pois apesar de um algoritmo de "Noise" permitir a criação de terreno de aspeto mais natural do que através da geração por "Heightmaps" terreno natural, como por exemplo uma montanha, varia de acordo com uma elevado número de fatores. A aplicação de agentes para ajudar a gerar terreno que possua múltiplas características é outra hipótese válida por oferecer um nível de controlo sobre o processo de geração do terreno que não é dada por Métodos de Fractais. Na geração de terreno baseada em agentes são tipicamente aplicados múltiplos agentes para que o terreno seja construido progressivamente por agentes especializados, fazendo com que o detalhe do terreno vá aumentando à medida que o tempo passa. É ainda possível recorrer-se a algoritmos de procura para tratar de casos em que a geração do terreno sofre de restrições relacionadas com a navegação no mapa.

Esta última abordagem fornece ainda mais controlo sobre a geração do que a geração por agentes, destacando-se Frade [26] na área da Geração de Terrenos por algoritmos de procura, implementando elementos de programação genética e algoritmos evolutivos. Antes do desenvolvimento desta aplicação, foram considerados os elementos que definem um algoritmo deste teor e quais os necessários para a obtenção de um Gerador de Terreno que se pode descrever como perfeito. São também tidas em conta múltiplas problemáticas, como por exemplo:

- Abordagens à implementação de *LoD* (ou "Level of Detail") na geração do mapa;
- *Técnicas de geração*, vulgarmente utilizadas;
- Os *benefícios e malefícios* provenientes das técnicas de programação a utilizar.

O gerador apresentado neste trabalho foi desenvolvido em MatLab com o auxilio da ferramenta GPLAB (sistema que facilita o desenvolvimento de Algoritmos Genéticos em MatLab) e permite que cada mapa gerado seja diferente do anterior graças à implementação de elementos genéticos. Note-se que após uma série de testes, durante a fase evolutiva do algoritmo de geração, o resultado final não é afetado pelo uso de um LoD baixo, uma vez que o LoD utilizado é, até certo ponto, independente do algoritmo de geração. Isto

dá azo a tempos de geração mais baixos e aumenta o nível de interatividade presente na ferramenta.

Frade refere que *"nos dias que correm muitas empresas de publicação e desenvolvimento de videojogos requerem que os jogadores tenham o mesmo tipo de experiência ao tomar as mesmas decisões num videojogo"* [26]. A geração automatizada de terrenos permite o controlo do mapa gerado a dois níveis, uma vez que, apesar da aleatoriedade envolvida na geração do mesmo, as características fundamentais do mapa gerado serão sempre as mesmas. Além disso, se existir a necessidade de manter o mesmo mapa ao longo de múltiplos jogos, é possível manter o mesmo valor aleatório (ou "seed") utilizado na geração do mapa, o que resulta, obrigatoriamente, em mapas iguais independentemente do número de vezes que o algoritmo de geração é executado.

Frade, no entanto, apresenta um único método para a geração de terreno. Atualmente, apesar dos "mundos" tridimensionais terem, pelo menos visualmente, aumentado o grau de complexidade, é possível verificar que as ferramentas de modelação utilizadas na sua criação não têm sido sujeitas ao mesmo tipo de processo evolutivo.

Apesar da geração procedimental ser um tópico de pesquisa já há alguns anos [32], esta não é tipicamente implementada na área da geração de terrenos, apesar de, com o tempo os resultados obtidos por estes métodos se terem vindo a tornar cada vez mais promissores.

À medida que estas técnicas vão sendo refinadas, o foco destes algoritmos passa a estar noutros aspetos da geração de terrenos, sendo o ponto de maior incidência original, o uso de Mapas de Alturas (ou HeightMaps) para a geração de altitude em terrenos planos, centrando-se no uso de métodos e algoritmos que podem ser implementados para facilitar e complementar a geração de terrenos e mapas, variando desde a geração de áreas urbanas a lagos e rios, passando inclusive por estabelecer redes de estradas que ligam cidades previamente geradas. Isto permite um maior detalhe nos elementos como, por exemplo a vegetação, tornando-se possível efetuar uma subdivisão dos elementos usados na geração detalhada de um mapa/terreno, bem como definir diversas formas de abordar pontos fulcrais na geração de cada um deles.

Graças a toda a pesquisa feita na área, viabilizou-se agora a utilização de "Blocos" que podem ser usados na geração de mapas cada vez mais complexos, permitindo assim um aumento significativo na quantidade de variações que podem ocorrer num mapa recém-gerado, pela simplicidade que estes trazem ao processo de gerar um novo terreno.

Um exemplo da aplicação de blocos à geração de terrenos é o Minecraft, um jogo do género Sandbox de imensa popularidade que conta com um gerador de terreno por blocos para criar um mundo com uma multitude de biomas de acordo com um conjunto de variáveis de ambiente. Nas versões anteriores à 1.8 do jogo, o tipo de bioma era gerado de acordo com certos elementos (como chuva e temperatura) cujos valores eram obtidos aleatoriamente [2]. O funcionamento das versões de Minecraft mencionadas é de particular interesse por se assemelhar ao de um Sistema de Regras, dando a entender que é possível

criar um gerador de terreno procedimental que funcione usando um SBR como base. Em versões posteriores à 1.8 passaram a ser atribuídos biomas específicos a fractais aleatórios do mapa para gerar biomas, sendo novos biomas introduzidos posteriormente para permitir que a transição entre biomas correspondentes a determinadas áreas do mapa seja feita de forma suave. Esta alteração deu-se para permitir uma integração mais fácil de novos biomas no jogo, sendo o sistema anterior descartado. O gerador de terreno do Minecraft é de particular importância a este projecto não só pela semelhança do seu gerador pré-1.8 a um sistema de regras, mas também pelo facto de todo o terreno ser gerado em Anytime. Ao contrario de jogos como o Age of Empires ou o Civilization, em que todo o mapa é gerado no inicio de uma partida, o Minecraft gera o mapa em torno do jogador de acordo com a sua posição.

2.4 Programação Dinâmica

A implementação de elementos de programação dinâmica é de interesse ao tópico em mãos por ser regularmente aplicada na área da Inteligência Artificial de modo a permitir que soluções futuras a um problema possam ser inferidas a partir de soluções previamente obtidas, algo de particular interesse neste projeto. Em 1993 Barto, Bradtke e Singh analisam formas de aplicar programação dinâmica a IA que funcione em tempo real [25], sendo que apesar o de artigo por eles publicado não ser particularmente recente, facilmente serve de referência devido ao detalhe e à informação nele contido. Segundo o grupo, algoritmos de programação dinâmica convencionais têm um nível de utilidade limitado no que toca a problemas cujo espaço de estados se verifique ser bastante alargado, como por exemplo problemas de IA em que o espaço de estados aumenta de forma combinatória, pois estes requerem que todo o espaço de estados seja expandido e o custo de cada estado posteriormente guardado em memória. O grupo sugere a utilização de pesquisa heurística devido à sua capacidade de explorar seletivamente o espaço de estados de um problema mantendo deste modo a eficiência do algoritmo.

O uso de programação dinâmica oferece no entanto uma vantagem sobre a pesquisa heurística, pois ao guardar os valores do custo numa estrutura de dados permanente, torna-se possível a aprendizagem do algoritmo fazendo com que os resultados obtidos se aproximem do resultado ótimo pretendido de forma cada vez mais rápida. Note-se que apesar de certos algoritmos de pesquisa heurística, como o A, atualizarem os valores de custo do espaço de estados a função heurística usada para a atribuição destes valores nunca é atualizada, fazendo com que, a não ser que os valores correspondentes a cada estado sejam guardados, não é possível conseguir um algoritmo com capacidade para “aprender”. Em termos de aprendizagem, o grupo considera que algoritmos de programação dinâmica que funcione offline são incapazes de ser algoritmos de aprendizagem genuínos por não serem desenhados com vista a serem aplicados durante a fase de resolução de problemas,

enquanto que a fase de aprendizagem decorre especificamente neste ponto. É no entanto possível recorrer à aplicação de um algoritmo de programação dinâmica sobre um outro algoritmo de modo a que o primeiro influencie e seja influenciado pelas decisões do segundo, considerando-se esta situação como um caso à parte utilizável em situações de resolução de problemas e de aprendizagem automática.

O sistema de monitorização de Anytime de Zilberstrin [36] anteriormente referido faz uso de elementos de programação dinâmica de modo a tentar inferir qual o tempo ótimo para a paragem do sistema Anytime, tentando garantir que o resultado dado pelo algoritmo Anytime monitorizado é o melhor possível. Este sistema tem em conta o custo da monitorização da solução e até que ponto é que a qualidade da solução pode ser estimada, permitindo deste modo avaliar o "trade-off" entre estes dois fatores. O grupo consegue desta forma construir um algoritmo com capacidade para gerar métodos de monitorização de Anytime que demonstrem ser superiores em termos de eficiência face a algoritmos de pesquisa heuristica, expandindo sobre trabalho préviamente realizado [30].

Capítulo 3

Análise

O objetivo principal deste projeto é tentar criar um algoritmo de decisão que faça uso de um sistema de regras e que se revele capaz de tomar decisões coerentes em Anytime.

Definiram-se para o efeito as seguintes questões de partida:

- Quais as ferramentas a usar no desenvolvimento desta aplicação?
- É possível implementar uma solução viável?
- O que torna a solução obtida viável?
- Qual o melhor género de jogo onde implementar esta solução?

As respostas a estas questões são apresentadas nas próximas secções.

3.1 Quais as ferramentas a usar no desenvolvimento desta aplicação?

Escolheu usar-se o motor de jogo Unity3d [17], versão 5 na altura da escrita deste documento, providenciado pela Unity Technologies, para desenvolver uma base ao sistema pretendido, por ser uma ferramenta de fácil utilização, rápida aprendizagem e por possuir uma interface simples de utilizar e detentora de uma biblioteca de recursos online. No entanto, a ferramenta apenas apresenta suporte nativo para Javascript e C#, sofrendo ainda de uma limitação a nível da versão de .Net que pode ser aplicada no desenvolvimento de jogos em C#, aceitando apenas a versão 2.0 da "framework", com alguns elementos adicionais integrados.

Tendo em conta os recursos mencionados, pretende-se criar a aplicação de exemplo no Unity (algo que a quantidade de recursos disponíveis facilita), seguindo-se o desenvolvimento de um sistema de regras simplificado que interaja com o exemplo através de uma classe que sirva de ponte entre o C# e este sistema.

A intenção inicial passava por fazer uso da linguagem de programação Python com o addon Pyke [9] para o desenvolvimento do sistema de regras. Todo o código relativo ao SBR a implementar seria escrito em Python e a interface com o C# do Unity seria estabelecida recorrendo à ferramenta IronPython [5]. Mais tarde percebeu-se que, estando estas ferramentas a cair em desuso (faltando suporte à sua utilização), o melhor a fazer seria fazer uso de uma ferramenta que consiga fazer o mesmo trabalho de forma mais simplificada, ao mesmo tempo que fornece um número de recursos mais extenso.

Pretendendo facilitar-se o desenvolvimento do código relativo ao sistema de regras, decidiu fazer-se uso de uma de duas linguagens criada especificamente para a implementação deste tipo de sistemas, CLIPS [1] e Prolog [12]. Ambas as linguagens têm como objetivo permitir ao utilizador desenvolver sistemas de regras, sendo o Prolog aquela que é mais comummente utilizada.

De modo a verificar qual destas ferramentas seria a mais viável para uma implementação em C# recorreu-se a uma bateria de testes simples para verificar a eficiência de cada uma.

Esta bateria de testes consistiu em efetuar os cálculos relativos ao vigésimo número da sequência de Fibonacci ($F_n = F_{n-1} + F_{n-2}$) e verificar quanto tempo é que cada linguagem demorava a obter o valor pretendido.

Tendo efetuado o mesmo calculo 50 vezes, obteve-se a seguinte média de resultados:

Tabela 3.1 Tempo gasto pelo CLIPS e pelo Prolog para obter $F(20)$.

CLIPS	Prolog
0.04699 s	0.02401 s

Tendo em conta os resultados obtidos, optou-se por implementar o sistema de regras em Prolog, e a interface a implementar na solução seria o tuProlog [15], uma ferramenta que, normalmente, serve para estabelecer uma interface entre as linguagens Java e Prolog, que tem vindo a ser adaptada à linguagem C#, e por ser compatível com a distribuição de .Net usada pelo Unity.

O Microsoft Visual Studio 2015 foi usado com o objetivo de testar certos elementos da implementação num ambiente exterior ao Unity. Esta IDE é eficiente, sendo utilizada por muitos developers que trabalham tanto com C# como com a framework .Net. O Visual Studio permite a criação e execução rápida de programas simples, sendo a versão da framework .Net sobre a qual o programa será executado facilmente alterável de modo a que o código seja compatível com a versão usada pelo Unity, fazendo desta IDE uma ferramenta extremamente flexível.

Tendo analisado as ferramentas disponíveis, passou-se ao design base da solução. A solução em si deve ser composta por 3 elementos, sendo estes o Gerador de Terreno, a Interface com o Prolog e o Sistema de Decisão que irá escolher qual o tipo de bloco a gerar. O tipo de decisões a efetuar pelo Sistema de Decisão implementado deve também

ter em conta três fatores base: a **Altura** no terreno em que o bloco a gerar se encontra, a **Temperatura** e a **Humidade** na altura da geração. Isto dará ao sistema uma maior variedade de blocos e permitir-lhe-á responder a situações mais complexas.

Sendo um "mundo" por blocos algo relativamente rudimentar, não houve necessidade de criar novos "Assets" gráficos através do uso de um programa de modelação 3D como por exemplo o Blender, uma vez que o Unity só por si oferece ferramentas de modelação básicas que servem o propósito do projeto.

No final do projeto, devido a uma multitude de problemas que foram surgindo, houve ainda a necessidade de se recorrer ao uso da biblioteca LINQ (Language-Integrated Query), presente na framework .Net desde a versão 3.0, por ser compatível com o Unity. Estes problemas e a razão do uso do LINQ serão mencionados posteriormente neste capítulo.

3.2 É possível implementar uma solução viável?

A resposta a esta questão tem-se revelado algo complexa, principalmente pela falta de recursos ligados à aplicação, nomeadamente pela falta de conceitos relacionados com Anytime ou SBRs a videojogos. No entanto, existem já jogos que funcionam usando conceitos relacionados com ambos os algoritmos, pelo que se acredita ser possível proceder à sua implementação num espaço de jogo, possibilitando assim a criação de um algoritmo que faça uso de um SBR e que consiga correr sobre um ambiente Anytime à medida que gera, consistentemente, respostas válidas aos pedidos feitos pelo sistema de jogo.

3.3 O que torna a solução viável?

A viabilidade da solução é garantida através da implementação de um algoritmo que possua as características básicas que definem um algoritmo Anytime. Este deve gerar novas regras após receber um número limitado de casos, quando lhe é pedido algo de novo, sendo que a base de conhecimento deve aumentar de cada vez que encontra uma situação nova, ou seja, uma regra que ainda não se encontra nela contida.

Isto garante ao algoritmo, no mínimo, a Monotonicidade, a Consistência e a Capacidade de melhorar que se encontram em algoritmos Anytime, pois com um conjunto diminuto de regras base, a qualidade da solução obtida será pior comparativamente a um caso em que o algoritmo foi inicializado com um conjunto de regras de dimensão superior ao primeiro. Isto permite também afirmar que, à medida que o algoritmo trata de casos novos, a sua base de conhecimento vai aumentando, resultando em respostas mais rápidas para casos conhecidos da aplicação. A Consistência do algoritmo advém do facto de tratar os mesmos conjuntos de dados da mesma forma, fazendo com que, independentemente do número de vezes que um pedido lhe seja feito, se os valores forem iguais em todas as ocasiões, o resultado será sempre o mesmo.

3.4 Qual o melhor tipo de jogo onde implementar uma solução deste género?

Jogos que pretendem oferecer aos jogadores um mundo infinito recorrem muitas vezes à geração procedimental de terrenos de modo a facilitar a criação de um espaço infinitamente expansível. Estes jogos, chamados jogos "Sandbox" fazem uso frequente de algoritmos de geração de terrenos para garantir que o jogador consegue explorar o mapa de jogo de forma indefinida. Um exemplo deste tipo de jogos onde já se fundem conceitos relacionados com Anytime e com SBRs é o Minecraft, um jogo que possui um sistema de geração que cria terreno de acordo com a posição do jogador. Esta situação é possível porque o terreno é subdividido em biomas que são gerados de acordo com um sistema de regras próprio.

Isto permite ter uma base para a criação de um algoritmo que sirva as mesmas funções que este gerador, tentando no entanto, diminuir o nível de aleatoriedade presente no gerador original do Minecraft, esperando garantir uma geração de terreno que não apresente variações de larga escala no mapa final. A principal diferença entre o gerador do Minecraft e o apresentado é o facto de o primeiro não ter capacidade para aprender a lidar com novos casos por si só, sendo que todos os casos possíveis têm que estar pré-programados no código do jogo.

3.5 Sumário

A informação relativa à implementação de algoritmos Anytime na área dos videojogos é escassa pelo que esta secção pretende simplificar os problemas impostos pelo desenvolvimento de uma aplicação Anytime na área dos videojogos. A falta de exemplos concretos em que SBRs são implementados sobre ambientes Anytime na área de videojogos dificulta a procura por uma solução viável ao problema em mãos, pois poucos são os jogos onde comportamentos que consolidam Anytime com SBRs são visíveis.

Visando encontrar uma solução viável ao problema em mãos foi definido um conjunto de ferramentas cujo principal objetivo é facilitar a criação de uma aplicação que vá de encontro ao pretendido, recorrendo-se para o efeito ao uso de um Motor de Jogo (Unity) e uma linguagem de programação que permita o desenvolvimento facilitado do sistema de regras usado pelo gerador de terreno (Prolog). Posteriormente, devido a complicações relacionadas com a interface entre a linguagem usada pelo Motor de Jogo escolhido (C#) e o Prolog, ficou decidido que o sistema de regras a integrar na aplicação seria desenvolvido unicamente em C# (detalhado no capítulo 5). A aplicação final pretende ser semelhante ao Minecraft, pois este possui um sistema de geração procedimental que pode ser adaptado de modo a servir os propósitos do projeto.

Capítulo 4

Desenho

Tendo sido feita a análise do problema em mãos passou-se ao desenho de uma solução viável, passando esta por criar um gerador de terreno por blocos em Unity semelhante ao do Minecraft que cria um "Mundo" construindo-o a partir de "Chunks" de 16 por 16 blocos, gerados apenas quando o jogador se aproxima de uma área vazia. Estes "Chunks" são removidos do "mundo" quando se afastam em demasia da posição corrente do jogador, garantindo a eficiência do jogo, sendo que estes conjuntos de blocos podem ser carregados posteriormente caso o jogador se aproxime novamente deles.

O "Mundo" onde os "Chunks" são representados é um espaço cartesiano onde são estes podem ser colocados e removidos, sendo o seu estado corrente reavaliando constantemente de modo a confirmar a posição do jogador para saber quando gerar ou remover "Chunks" do espaço. A criação de "Chunks" deve ser feita de modo a que os blocos não tenham alturas iguais, havendo a necessidade de usar um algoritmo de "Noise". Este algoritmo é usado para a atribuição de um valor de altura a componentes de mapas planos, sendo por este motivo fundamental na geração de mapas tridimensionais de forma procedimental, podendo também ser aplicado à geração de efeitos de fogo e àgua, bem como à geração de nuvens.

O código C# deverá interagir com uma interface e efetuar pedidos ao Prolog contendo os valores das variáveis pertinentes (Altura, Temperatura e Humidade) devendo o Prolog responder com o Bloco escolhido. O Prolog atribui também novos valores para a Temperatura e Humidade, provocando ligeiras alterações no ambiente que resultarão na escolha posterior de novos blocos.

A secção do projeto desenvolvida em Prolog representa o cérebro da aplicação, recebendo apenas um conjunto de dados que representem o estado corrente do "mundo" de forma a poder decidir qual o tipo de terreno a gerar, por se pretender a criação de um "mundo" com uma aparência natural, alterando para o efeito as variáveis correspondentes aos dados recebidos após a geração.

Com base nesta informação é possível desenhar uma versão simplista do sistema semelhante ao descrito pelo diagrama 4.1, note-se que para cada um dos diagramas

apresentados neste capítulo, cada elemento presente corresponde um "Asset" do Unity, ou seja, qualquer item que possa fazer parte do jogo ou projeto [18]. Neste caso os Assets em questão encontram-se divididos em Scripts, Prefabs [21] e GameObjects [19], sendo cada elemento no diagrama um Script a não ser que seja especificado o seu tipo (Prefab ou Object). Note-se ainda que estes diagramas representam meramente o desenho inicial do sistema, sendo que o Diagrama de Classes do sistema se encontra em anexo.

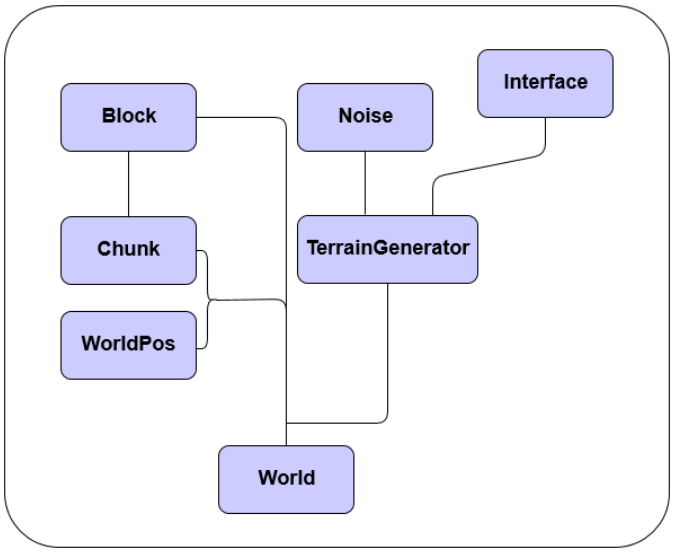

Figura 4.1: Desenho da Aplicação Inicial.

Durante o desenvolvimento da aplicação houve a necessidade de adicionar elementos que não tinham inicialmente sido tidos em conta, como por exemplo a premência de se recorrer a serialização (Serialization e Save) para garantir que uma área do terreno gerado pudesse facilmente ser acedida e recriada após a sua destruição em vez que ser gerada novamente a partir do zero. Foi também necessário criar uma entidade que represente o jogador ("Player") para que seja gerado novo terreno apenas quando necessário (sendo para o efeito usado o script LoadChunks), garantindo assim a Interruptibilidade e Capacidade de recuperação de interrupções presentes nos algoritmos Anytime, pois a geração não é feita de forma continua mas sim apenas quando o jogador se aproxima de uma área onde ainda não existe terreno gerado.

Foi ainda adicionado um Objecto de jogo que consiga representar o mundo gerado, neste caso o World (Object), e um elemento serializável prefabricado que sirva como base à criação de novos "Chunks", o Chunk (Prefab). Estas alterações resultaram num sistema descrito pelo diagrama demonstrado na figura 4.2.

Foi criado o script adicional "MeshData" para facilitar a criação de múltiplos blocos a partir de um script de bloco genérico, conseguindo ainda tornar o algoritmo de geração mais eficiente através da remoção de áreas de jogo que não conseguem ser vistas pelo jogador.

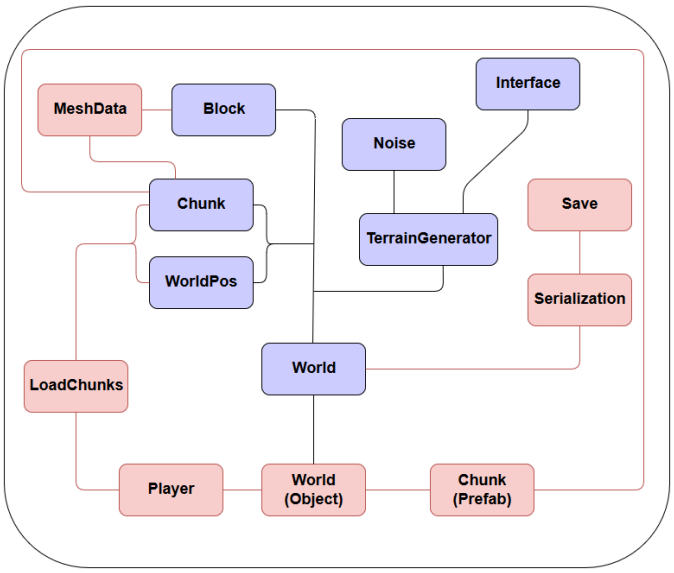

Figura 4.2: Novo Desenho da Aplicação.

Tentando inicialmente refazer a interface em si, recorreu-se à biblioteca de integração de Prolog em C# disponibilizada pelo SWI-pl [12] [13]. Apesar desta nova biblioteca resolver o problema originalmente causado pelo tuProlog, surgiu a questão da incompatibilidade entre as versões de .Net usadas. Estes problemas encontram-se descritos em melhor detalhe no capítulo 5.

Estes contratempos conduziram à criação de um algoritmo de decisão e inferência criado exclusivamente em C#, com as mesmas capacidades que se pretendiam do código de Prolog, conseguindo-se inferir novas regras, com base no conhecimento que inicialmente lhe é passado, tornando-o, efetivamente, mais eficiente ao longo do tempo. O resultado final é um sistema como aquele que se apresenta no diagrama da aplicação da figura 4.3:

Este novo algoritmo está dividido em duas secções: um Sistema de Regras (“RuleSystem”) e uma estrutura de dados usada para o armazenamento de valores (“Attribute”). O Sistema de Regras possui um leitor de XML desenhado especificamente para ler ficheiros XML onde se encontram presentes os valores que irão definir o estado inicial do ambiente gerado, bem como um conjunto de factos e regras correspondentes a uma amostra inicial que permitirão ao algoritmo ter uma base de conhecimento que este pode usar como ponto de partida para inferir os resultados de casos cujas soluções ele desconhece. As regras presentes na tabela recorrem aos atributos correspondentes à Altura do bloco,Temperatura e Humidade para determinarem o tipo do novo bloco a gerar. Estas regras têm o seguinte formato:

No inicio do processo de execução o algoritmo lê o ficheiro XML, extraindo os valores correspondentes ao estado inicial do mundo a gerar, interpretando-os e alocando-os às suas variáveis correspondentes. Posteriormente são obtidos os factos que são alocados a estruturas de dados do tipo “Attribute” em conjunto com as regras que são adicionadas a

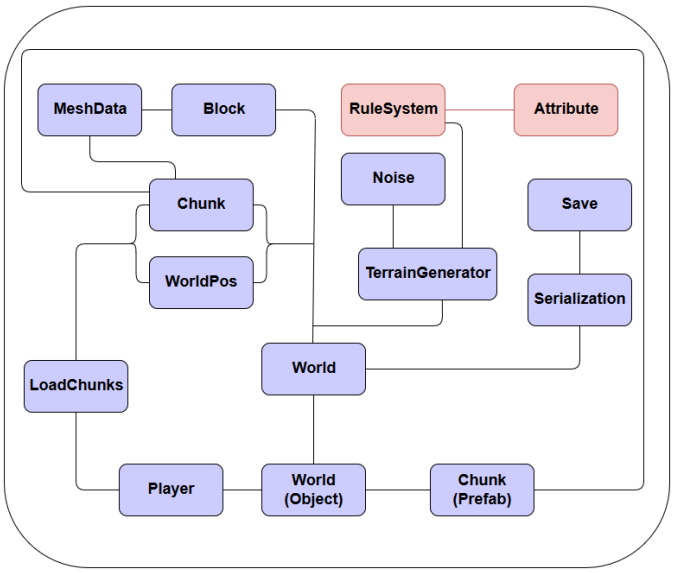

Figura 4.3: Desenho da Aplicação Final.

Algoritmo 2 Exemplo de uma Regra a aplicar

if $Height = Med$
and $Temp = Low$
and $Humid = Med$
then $Block = Snow$

uma tabela de dados, que podem ser facilmente acedidas por LINQ [6], uma biblioteca fornecida pela Microsoft que permite efetuar queries semelhantes às usadas em SQL num ambiente .NET, permitindo assim um acesso rápido às regras armazenadas.

Esta solução permite uma divisão clara das componentes que compõem a IA do algoritmo, fornecendo a potenciais utilizadores formas de alterar o resultado final da geração sem ser necessário efetuar alterações ao comportamento do algoritmo.

4.1 Sumário

A fase de desenho tem como objetivo simplificar o processo de desenvolvimento da aplicação base através da análise e estruturação das componentes que a versão final do projeto irá necessitar.

Com a evolução do projeto ocorreram múltiplas alterações ao desenho inicial da aplicação, sendo que as principais alterações estruturais ao código ocorreram ao longo do processo de desenvolvimento. Durante o desenvolvimento da aplicação houve a necessidade de adicionar vários elementos, na altura em falta, ao gerador de terreno para garantir que este conseguia funcionar de acordo com o que era expectável.

Posteriormente, após diversas tentativas de implementação da interface entre C# e Prolog, foi tomada a decisão de implementar o sistema de regram em C#, resolvendo desta

forma todos os problemas inerentes ao uso do Prolog (detalhado no capítulo 5).

Graças a esta estruturação do código foi possível acelerar o processo de desenvolvimento e debug da aplicação final, tendo este processo sido imperativo na obtenção de um programa que vai de encontro aos objetivos propostos.

Capítulo 5

Implementação

Ao longo do desenvolvimento do projeto surgiram problemas que tiveram de ser resolvidos através de algumas alterações drásticas ao código implementado. Esta situação provocou atrasos que afetaram diretamente a qualidade final do código implementado, resultando em múltiplas alterações a código antes considerado estável.

Foi também criado um Gerador de Terreno Simplificado recorrendo quase exclusivamente às ferramentas providenciadas pelo Unity 5, tendo sido usadas ainda texturas externas à aplicação como complemento ao novo gerador. O objetivo principal era comparar a velocidade de ambos os geradores. Os resultados obtidos são discutidos no próximo capítulo.

5.1 Gerador de terreno

Nesta secção serão descritos os scripts principais que compõem o primeiro gerador de terreno implementado, procedendo-se detalhadamente à descrição do seu funcionamento, tendo o cuidado de realçar eventos que tenham tido impacto no desenvolvimento do projeto.

Com o objetivo de facilitar o desenvolvimento desta fase da aplicação recorreu-se a um tutorial que visa permitir a geração de um terreno por blocos [24]. Graças a este tutorial foi possível adquirir a noção de elementos que não estavam presentes no plano original, sendo que a implementação da aplicação difere do diagrama inicial, estando o novo diagrama representado na figura 5.1.

5.1.1 Scripts

Block

Cada um destes blocos é uma estrutura serializável, representada por um conjunto de faces, sendo que cada uma delas é, por sua vez, composta por dois triângulos. Todos os métodos presentes na classe "Block" são virtuais, permitindo que estes sejam "overriden" de modo a garantir que a criação de novos blocos é o mais fácil possível, por

exemplo, as classes “BlockAir” (um bloco invisível), “BlockSnow” (um bloco de neve) ou o “BlockGrass” (um bloco com relva), são extensões da classe base “Block”.

Para tratar da representação de um bloco no “mundo” é usado um atlas de texturas (elementos que definem o aspeto visual de um objeto tridimensional), como o representado na figura 5.2, de onde é lido qual o segmento que vai ser usado para a criação do bloco. Esta informação é guardada numa estrutura de dados Tile sob a forma de uma posição num plano bidimensional (x,y).

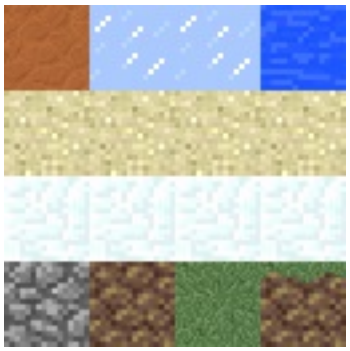

Figura 5.1: Atlas de Texturas.

Posteriormente é aplicada uma textura sobre as faces de acordo com a sua posição (x,y) num “Atlas de Texturas”. Por exemplo, um bloco de pedra, que será igual em todas as faces, terá cada uma das suas faces a apontar para as coordenadas (0,0) no atlas, enquanto que para um bloco com relva, a face inferior apontará para (1,0), a superior para (2,0) e as restantes para (3,0).

Um bloco tem também a capacidade de verificar se cada uma das suas faces deve ou não ser sólida ao confirmar se os blocos que lhe são vizinhos são, ou não, sólidos, e, caso o bloco seja sólido, a face não será renderizada. A distinção entre faces, é conseguida usando um enumerador que representa cada um dos pontos cardeais, com a adição de dois valores que representam o topo e o fundo do bloco, permitindo distinguir facilmente uma face de outra. Na figura 5.3 apresenta-se o resultado da geração de um bloco base.

MeshData

De modo a representar as texturas de cada face de um bloco deve atribuir-se-lhe uma “Mesh” que será posteriormente renderizada no espaço cartesiano pelo motor de jogo. Isto é conseguido através do uso de um script que contém os dados correspondentes aos triângulos usados na criação de um bloco (gerados através de um conjunto de vértices e agrupados numa lista de objetos “Vector3s” que contêm as posições no espaço tridimensional), como se apresenta na figura 5.4.

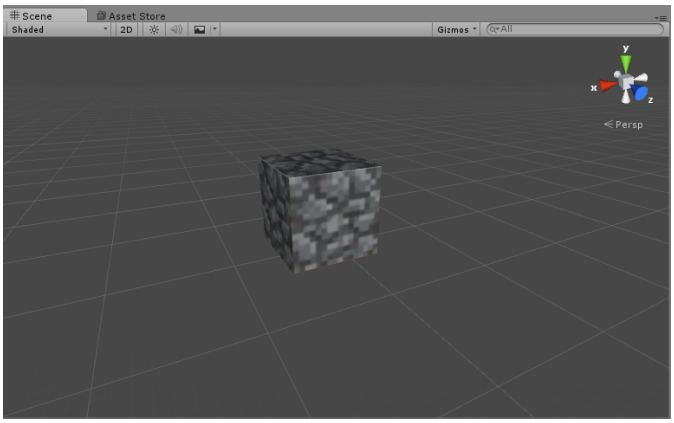

Figura 5.2: Bloco Base.

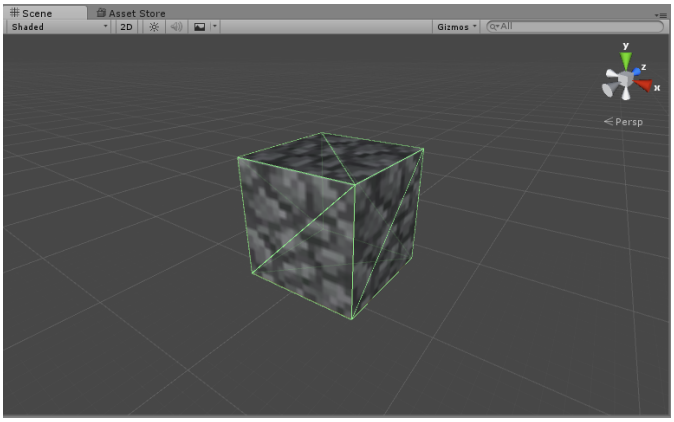

Figura 5.3: Mesh Collider sobre um Bloco Base.

Chunk

Enquanto os blocos servem de base à aplicação, os "Chunks" são usados para formar um conjunto de 16^3 blocos. O objetivo desta ação é manter o gerador de terreno eficiente, tentando evitar que este esteja constantemente a apagar e gerar blocos novos.

Um objeto do tipo "Chunk" é constituído principalmente por um array tridimensional de blocos ($Block[,,]$) com uma dimensão definida de acordo com o valor da variável "ChunkSize" (que neste caso em particular é 16), sendo esta lista usada para definir qual a posição cartesiana de um bloco que pertence ao "Chunk". Um "Chunk" consegue interagir com um bloco de modo a obter os blocos que lhe são adjacentes, verificando inclusive se contém ou não um bloco com coordenadas determinadas na sua listagem de blocos.

Esta classe estende o "MonoBehaviour" [20], uma classe própria do Unity que deve ser estendida para permitir a integração do script nos objetos pertencentes ao espaço de jogo ("GameObjects" [19]), havendo a necessidade de recorrer a métodos próprios do Unity (como por exemplo o "Update").

Este script verifica de forma constante (a cada frame) se existe a necessidade de efetuar modificações no próprio "Chunk" através do método "Update", nativo ao Unity.

Se for esse o caso (ou se o "Chunk" tiver acabado de ser gerado é criada uma "Mesh") são "renderizados" apenas os lados externos do "Chunk" (correspondentes aos lados dos blocos que são de facto visíveis), o que torna a geração eficiente na medida em que não é necessário tratar a renderização de cada lado de um bloco de forma individual. Por exemplo, numa situação em que um "Chunk" fosse gerado e em que todos os 16^3 blocos estivessem presentes, existiria a necessidade de renderizar a "Mesh" de cada uma das faces de cada um dos cubos, algo que ocuparia uma quantidade desnecessariamente grande de memória.

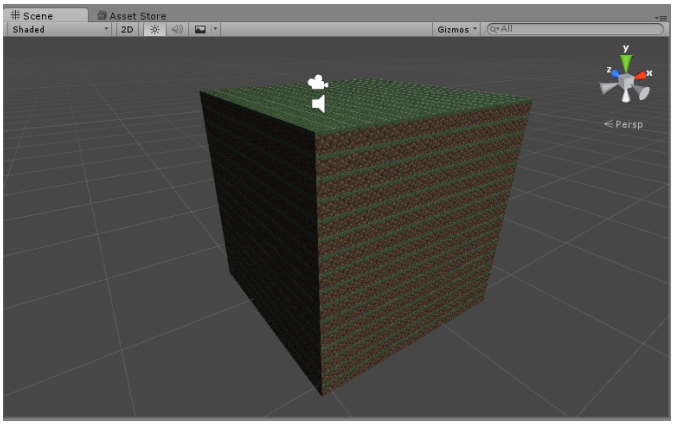

Figura 5.4: Chunk Básico.

WorldPos

O "WorldPos" é uma estrutura serializável simples que serve para ajudar a estabelecer qual a posição de um bloco ou "Chunk" no "mundo" gerado. Esta estrutura tem apenas um construtor que recebe como parametros posições nos eixos x,y e z, bem como um método que permite efectuar a comparação entre duas destas estruturas.

Save

O script Save guarda todos os blocos de um "Chunk" numa estrutura de dados do tipo $Dictionary < K, V >$ onde K representa uma Chave e V um valor associado a essa chave. Querendo indexar os blocos de acordo com a sua posição no espaço, K será do tipo WorldPos e V o Bloco que corresponde a essa posição. Isto permite que um "Chunk" seja guardado de forma fácil e rápida, para que possa ser carregado quando necessário.

Serialization

O script Serialization faz uso do Save para gravar um "Chunk" de blocos num ficheiro guardado dentro de uma pasta gerada na altura da criação do "mundo" do jogo. Estes "Chunks" são serializados antes de serem gravados de modo a permitir a sua escrita para

um ficheiro binário, sendo posteriormente desserializados na altura em que os ficheiros são lidos com o objectivo de regerar os "Chunks".

Noise

Esta classe representa um algoritmo de "Noise" tipicamente usado para gerar relevo em terrenos. Neste caso específico escolheu-se uma implementação do algoritmo Simplex Noise desenvolvido por Heikki Törmälä [11].

Apesar de o próprio Unity fornecer um algoritmo deste género (Perlin Noise), o Simplex Noise é mais eficiente do que o Perlin Noise, requerendo menos cálculos para gerar o mesmo tipo de terreno, com a vantagem adicional de conseguir gerar terrenos de aspeto mais natural do que o Perlin Noise, pois o resultado gerado é distribuído por hexágonos no Simplex Noise, em vez de quadrados, como acontece no Perlin Noise.

TerrainGen

O TerrainGen trata da geração do terreno propriamente dita, contendo no seu interior um método que é usado para atribuir uma altura a cada bloco de um "Chunk", definindo ainda se o bloco a gerar é ou não sólido, usando o resultado do script de SimplexNoise.

A função ChunkGen gera um "Chunk" por coluna de blocos, tal como mostra a figura 5.5, usando o valor do Noise para determinar a sua altura. Posteriormente a altura do bloco resultante é comparada com um conjunto de valores que representam pontos a partir dos quais a altura passa a afetar o tipo de bloco de forma diferente, criando um BlockAir (um bloco invisível) caso a sua altura supere a obtida pelo algoritmo.

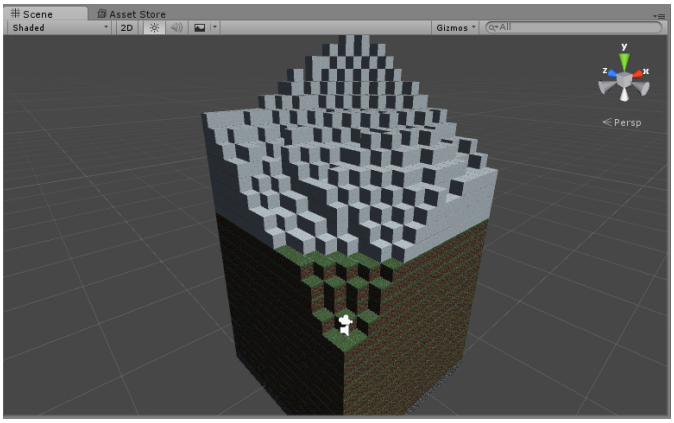

Figura 5.5: Chunk gerado pelo TerrainGen.

World

Este script permite definir o "mundo" no qual serão gerados os "Chunks" de blocos. De forma a permitir esta geração, define-se este script como uma extensão de um MonoBehaviour de modo a ser adicionado a um GameObject para que possa fazer uso dos

métodos Start (que permite que um determinado segmento de código seja corrido sobre o objeto quando o jogo é inicializado) [22] e Update (que executa o código nele contido uma vez por cada frame) [23].

Os "Chunks" presentes neste "mundo" estão organizados de acordo com a sua WorldPos num $Dictionary < K, V >$ onde K representa uma Chave e V um valor associado a essa chave (neste caso K é um WorldPos e V um "Chunk"), sendo estes efetivamente gerados a partir de clones do "Prefab" "Chunk".

Este script trata da criação de "Chunks" recebendo as coordenadas que correspondem a um único vértice do "Chunk", sendo posteriormente instanciado um "Chunk" com inicio nessa posição e gerados os blocos que o compõem através de uma chamada à classe TerrainGen. Inversamente, a destruição de um "Chunk" (também definida no script) tenta obter um "Chunk" pré-existente a partir de uma posição no espaço cartesiano, usando o script Serialization para gravar o "Chunk" a destruir, eliminando-o do espaço de jogo e removendo-o do dicionário onde se encontram indexados todos os "Chunks" instanciados no "mundo". Para possibilitar a criação e alteração de "Chunks", também estão definidos métodos de "get" e "set" para blocos individuais.

Os acessos e alterações a blocos são feitos usando, principalmente, o método de acesso ao "Chunk" para obter o "Chunk" onde o bloco pretendido se encontra, procurando o bloco no interior do "Chunk" e retornando-o caso o encontre, se tal não acontecer o método devolve um bloco vazio (BlockAir). No caso do método que faz a alteração ao bloco no "Chunk", procura-se a posição onde deve ser feita a alteração ao "Chunk", definindo-se o tipo de bloco pretendido nessa posição.

Este script será implementado no World (Object) de modo a permitir a criação de novos "Chunks" no espaço de jogo.

LoadChunks

O LoadChunks é o script que estende o MonoBehaviour e será adicionado ao GameObject Player, estando encarregue da criação e remoção dos "Chunks" propriamente ditos. A cada frame é avaliada a posição do jogador, verificando-se se este se afastou ou aproximou demasiado de um "Chunk" previamente gerado ou ainda por gerar.

De acordo com a posição obtida o script verifica se existem "Chunks" que estejam a uma distancia muito grande do jogador, eliminando do mapa "Chunks" para os quais isto seja verdade. Feito isto, verifica se existem "Chunks" que deveriam ser gerados na área em redor do jogador, adicionando "Chunks" por gerar a uma lista usada para pedir ao script "World" para gerar o novo "Chunk". Este "Chunk" é então adicionado a uma updateList (uma lista de "Chunks" usada para conter "Chunks" presentes no "mundo" de jogo) que verifica se novos potenciais "Chunks" devem, ou não, ser gerados, sendo gerados apenas quando não estão presentes na lista.

Este script tem que ser colocado no objeto que representa o jogador, de modo a ter

sempre acesso à sua posição independentemente do movimento do mesmo, possibilitando desta forma a geração de novos "Chunks" à medida que o jogador se move pelo mapa. É deste modo possível implementar os elementos de *Interruptibilidade* e *Capacidade de recuperação de interrupções* presentes nos Algoritmos Anytime.

Player

O objeto correspondente ao jogador provém de um conjunto de scripts e "GameObjects" providenciados pelo Unity, que podem facilmente ser importados a partir do menu Assets >Import, selecionando a opção "Character Controler". Isto importará um conjunto de scripts editáveis que contêm código para um "GameObject" que pode ser usado com a finalidade de criar um elemento com visibilidade na primeira ou terceira pessoa, que é controlável pelo jogador.

Na aplicação de exemplo, o tipo de jogador escolhido vê o "mundo" do ponto de vista da primeira pessoa, sendo que lhe é adicionado o script LoadChunks para garantir que o "mundo" é gerado de acordo com a sua posição.

5.1.2 Prefabs & GameObjects

Chunk (Prefab)

O "Prefab" "Chunk" é um objeto pré-fabricado que contém o script "Chunk", sendo facilmente gerável e instanciável no espaço tridimensional. Este prefab é usado para gerar terreno através de chamadas feitas a partir do script World.

World (Object)

O World (Object) é um GameObject simples que é adicionado à hierarquia e objetos de jogo do Unity. Este objeto é utilizado para representar o "mundo" gerado, usando apenas o script World e recebendo o "Prefab" do "Chunk" que este script requer para funcionar na sua plenitude.

5.1.3 Gerador de Terreno Simplificado

De modo a possibilitar a análise da eficiência do gerador de terreno inicialmente desenvolvido, foi criada outra aplicação, no Unity, que faz uso, principalmente, das funcionalidades disponibilizadas pelo Motor de jogo.

Esta versão do gerador é algo extremamente simples, sendo composto apenas por um único script cuja função principal é gerar segmentos de terreno, bloco a bloco, usando apenas os "prefabs" destes blocos.

Neste caso, em vez de ser usado apenas um bloco base para criar novos blocos, são criados três "Prefabs" que assumem a forma de um cubo simples com uma única textura.

O algoritmo de "Noise", usado na geração procedimental de relevo em mapas, aqui aplicado é o providenciado pelo Unity (Perlin Noise) e permite que o tipo de bloco seja definido de acordo com a altura a que se encontra. Por exemplo, um bloco gerado será de Neve se a altura for superior a um valor predefinido, Pedra se se encontrar abaixo de um limite inferior e Relva se estiver entre ambos os limites.

O Gerador está desenhado para seguir a posição do jogador e pode adicionar ou remover blocos de acordo com a sua posição, garantindo assim que o terreno gerado é efetivamente infinito.

5.2 Prolog

O código Prolog inicial foi desenhado de modo a conseguir desempenhar no mínimo três funções básicas necessárias ao funcionamento do algoritmo híbrido. São estas:

1. Oscilar internamente os valores das variáveis de ambiente;
2. Permitir a adição e remoção de novos Blocos e Regras;
3. Avaliar os valores de ambiente e decidir qual o bloco a retornar;

O Prolog desempenha as funções acima descritas através de conjuntos de regras específicos para a oscilação de valores e tratamento de factos.

A oscilação de valores consegue-se através do uso de um valor aleatório de zero a cem (gerado do lado do Unity) e de um valor que determina qual a probabilidade de ocorrerem alterações no valor a oscilar, sendo que a oscilação destes valores ocorre apenas quando o valor aleatório se encontra dentro do intervalo que determina se a variável de ambiente deve ou não sofrer oscilações.

Estes valores devem também oscilar de acordo com um valor mínimo e um valor máximo (definidos pelo utilizador), nunca podendo ultrapassar estes limites.

A escolha do tipo de bloco é feita de acordo com os factos presentes na base de factos do Prolog e o valor das variáveis que são passadas ao mesmo. Por exemplo, a escolha de um bloco com base na sua temperatura é feita analisando o valor da temperatura inicial, tentando encontrar uma regra em que o valor se enquadre, retornando o bloco pretendido, caso possível, e um bloco base (um bloco de pedra), caso o valor não se enquadre em nenhuma das regras.

Note-se que este código contém apenas um conjunto de regras que visam tratar da avaliação dos dados que lhe são passados pela interface, sendo que, novos factos e regras são adicionados através da própria interface.

Foram feitos testes ao código Prolog usando a IDE disponibilizada pelo tuProlog, que contém uma aplicação que facilita o desenvolvimento, execução e debug de código em Prolog.

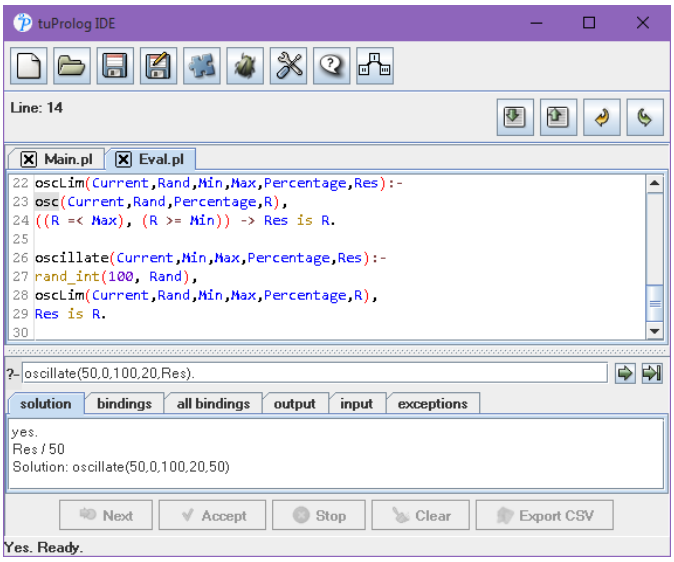

Figura 5.6: IDE disponibilizada pelo tuProlog.

Procedeu-se a uma implementação da interface inicial entre C# e Prolog fora do Unity, sendo esta montada sobre uma solução gerada automaticamente pelo Visual Studio 2015.

A partir deste procedimento foi possível verificar quais os pré-requisitos necessários ao funcionamento da interface bem como adquirir uma ideia das ferramentas disponibilizadas pela "Framework" do tuProlog e fazer testes preliminares ao comportamento do algoritmo.

A interface permitiu a adição e remoção de novos factos e regras pertinentes à aplicação de teste, especificamente a adição de novos blocos e respetivas condições, através da implementação de métodos que passam instruções de "Assert" (adição de novos factos) e "Retract" (remoção de factos) para o Prolog, recebendo como argumentos strings que representam os valores presentes nos factos a adicionar/remover. Estando estas funções desenhadas especificamente para serem implementadas sobre a aplicação de exemplo, o tipo de factos que podem ser adicionados e removidos da base de factos é restrito.

Para além da adição e remoção de factos existentes na base de dados do Prolog, a interface apresenta também a capacidade de efetuar pedidos ao código Prolog, contendo os valores de ambiente, que serão avaliados de acordo com as regras presentes no código, devolvendo o tipo de bloco a construir sob essas condições.

Posteriormente, os problemas resultantes do funcionamento da interface fornecida pelo tuProlog (sobre os quais discutiremos na próxima secção), levaram à necessidade de procurar uma forma de criar uma nova interface cuja sintaxe não fosse muito diferente da fornecida pelo tuProlog.

Tendo-se testado a interface entre Prolog e C# utilizando a biblioteca fornecida pelo SWI-pl no Visual Studio, procedeu-se à reimplementação da interface previamente construida usando o tuProlog.

Esta interface possui as mesmas características que aquela que foi feita usando o

tuProlog, com a vantagem de ser mais rápida.

5.3 Algoritmo de Decisão e Inferência em C#

De modo a replicar alguns dos aspetos básicos presentes no Prolog, foi feito um algoritmo de decisão e inferência com capacidade para ler um conjunto de factos e regras a partir de um ficheiro XML e aprender por si próprio quando encontra uma situação que não está presente na listagem de regras que ele possui na altura. Neste mesmo ficheiro XML é possível alterar e adicionar regras que deverão constar na base de regras do algoritmo, sendo que a adição de novos factos requer também a criação de uma variável do tipo "Attribute", da mesma forma que a edição dos intervalos presentes em cada facto requerem também a alteração dos valores que compõem o Enum "Values" (estrutura usada para converter os dados obtidos pelo XML para dados utilizáveis pelo algoritmo). Neste caso específico os valores que podem ser assumidos por um atributo na Base de Regras são Low, Med e High.

5.3.1 Attribute

A estrutura de dados "Attribute" é um script simples que serve unicamente para armazenar números inteiros e convertê-los para valores aceites pelo sistema de decisão. Isto é conseguido fazendo corresponder a cada atributo uma lista de valores que define os limites, permitindo decidir a que intervalo é que este valor inteiro pertence.

Por exemplo, se $0 = Low$ e $15 = High$, 10 será um valor entre estes dois limites, sendo por isso definido como um valor Médio ($10 = Med$).

```
<Fact id="Height">
    <Val id = "Min"  value= "-1024" />
    <Val id = "Low"  value= "0" />
    <Val id = "High" value= "15" />
    <Val id = "Max"  value= "1024" />
</Fact>

Attribute Height :
Integer Values = {-1024,0,15,1024}
Enum Values = {Min,Low,High,Max}
Height (10) = Med (Between Low & High)
```

Figura 5.7: Exemplo da atribuição de valores.

Esta conversão é importante por possibilitar uma comparação direta de valores com os atributos presentes na base de dados enquanto permite que seja usada uma sintaxe percetível pelo utilizador na definição do XML.

5.3.2 DecisionSystem

Este algoritmo usa DataTables para gerar tabelas de dados que serão posteriormente acedidas através de queries feitas por LINQ. Cada coluna da tabela corresponde a um tipo de dado (Temperatura, Humidade, Altura), sendo que o seu valor é dado de acordo com um Enumerador que determina se este valor é Baixo, Médio ou Alto.

A ideia de fazer uso desta estrutura de dados adveio da implementação do algoritmo ID3 [3] para a geração de árvores de decisão binárias, onde se usa uma "DataTable" para passar ao sistema a amostra inicial com a qual a árvore de decisão foi gerada.

O ficheiro XML contém um conjunto de factos e regras que são lidos na altura da inicialização do algoritmo. Os factos serão usados para criar uma listagem que contém valores iniciais a atribuir às variáveis que descrevem o ambiente a gerar e os conjuntos de intervalos presentes em cada estrutura da classe "Attribute", que servirão para determinar qual o bloco a usar. Em contrapartida, as regras alocadas a uma "DataTable" permitem que estas sejam armazenadas e acedidas durante o "runtime" da aplicação, sendo cada atributo principal (Temperatura, Humidade, Altura) guardado em colunas específicas.

Ao apresentar-se um pedido ao algoritmo, é realizada uma query em LINQ à tabela onde estão contidas as regras, sendo cada elemento passado em conjunto com o pedido convertido para um elemento secundário, o que permite a sua verificação de acordo com os dados que estão presentes na tabela.

Por exemplo, se a tabela de regras contiver as seguintes regras:

Tabela 5.1 Exemplo de tabela de regras

Altura	Temperatura	Humidade	Bloco
Med	Med	Med	Grass
High	Med	Med	Snow
Med	Low	Med	Snow

É inicialmente feita a seguinte query:

Tabela 5.2 Exemplo de uma Query a uma regra presente na tabela

Altura	Temperatura	Humidade	Bloco
Med	Low	Med	?

Ao encontrar a regra pretendida é retornado o Bloco correspondente à regra. Ou seja, se fosse feita uma query que procurasse o resultado para o conjunto de condições: $Altura = Med; Temperatura = Low; Humidade = Med$, o algoritmo iria devolver $Snow$ como demonstrado pela tabela 5.3, pois a regra estaria presente na tabela.

Num caso em que seja feita a query da tabela 5.4, ou seja, para o conjunto de condições: $Altura = High; Temperatura = Low; Humidade = Med$, não será possível devolver um tipo de bloco de imediato, pois não existe uma regra presente na tabela que corresponda ao que foi pedido na query.

Tabela 5.3 Resultado de uma Query a uma regra presente na tabela

Altura	Temperatura	Humidade	Bloco
Med	Med	Med	Grass
High	Med	Med	Snow
Med	Low	Med	Snow

Tabela 5.4 Exemplo de uma Query a uma regra não presente na tabela

Altura	Temperatura	Humidade	Bloco
High	Low	Med	?

Não existindo então esta regra, proceder-se-á a uma nova pesquisa usando os critérios presentes na tabela 5.5, que correspondem ao uso de apenas dois dos atributos instanciados. Este processo é executado sempre que é feito um pedido ao algoritmo que contenha uma regra que não esteja presente na tabela de regras.

Tabela 5.5 Query Secundária

Altura	Temp	Bloco
High	Low	?

OR

Temp	Humid	Bloco
Low	Med	?

OR

Altura	Humid	Bloco
High	Med	?

Numa implementação inicial do algoritmo, este limitava-se a verificar a lista de resultados retornada por esta query, respondendo ao pedido feito com o primeiro resultado encontrado, o que fazia com que pedidos que não tivessem uma regra presente na tabela inicial de regras resultassem em valores inconsistentes e, por vezes, até mesmo errados.

No sentido de corrigir estes erros, o algoritmo foi ajustado, alterando a query feita de modo a garantir com que a decisão tomada pelo algoritmo passe a ter em conta todos os resultados presentes na tabela, de modo a que se a procura não der resultados positivos, é feita uma nova query emparelhando cada um dos atributos individualmente e verificando quais os resultados possíveis, retornando o bloco mais comum de entre os blocos obtidos.

No caso apresentado, o resultado mais comum seria Snow (ou um bloco de neve), pelo que é esse o bloco devolvido pelo algoritmo, demonstrado pela tabela 5.6.

Como esta regra não existia, ela é posteriormente adicionada à tabela de regras (como exemplificado na tabela 5.7) de modo que o resultado pretendido possa ser obtido mais rapidamente. Se com este procedimento o resultado obtido não for válido, será colocado um bloco base (de pedra) por omissão.

Se uma query com dois parâmetros faz a tabela devolver mais do que um resultado,

Tabela 5.6 Resultado da Query Secundária.

Altura	Temp	Humid	Bloco
High	Low	**-**	**X**
High	**-**	Med	**Snow**
-	Low	Med	**Snow**

Tabela 5.7 Exemplo de adição de uma nova regra à tabela de regras.

Altura	Temperatura	Humidade	Bloco
Med	Med	Med	Grass
High	Med	Med	Snow
Med	Low	Med	Snow
High	Low	Med	Snow

todos eles serão tidos em consideração para a decisão final, criando-se uma lista de potenciais resultados em que o tipo de Bloco mais comum de entre os resultados obtidos está em primeiro lugar (processo executado automaticamente pelo LINQ). É posteriormente selecionado o primeiro elemento da lista e adicionada a nova regra à tabela de regras. Se estiverem presentes múltiplos tipos de blocos em igual número na lista, será escolhido o primeiro elemento de entre todos os resultados presentes na listagem.

Para demonstrar a evolução do algoritmo de acordo com o tempo apresenta-se uma lista com a evolução da tabela de regras ao longo do tempo, em que ”Count“ representa a iteração em que foi feita a adição da nova regra à tabela. Neste caso, os valores numéricos (1, 2 e 3) representam os valores Low, Med e $High$ respetivamente, seguindo-se o tipo de bloco que corresponde às condições definidas pelas regras. Note-se que as colunas das tabelas apresentadas na figura 5.8 estão ordenadas da mesma forma que as tabelas anteriormente apresentadas.

De modo a possibilitar testes ao algoritmo, criou-se uma pequena aplicação independente, totalmente em C#, usando o Visual Studio. Graças a esta implementação parcial foi possível afinar o processo de decisão tornando-o mais natural.

Com a implementação deste algoritmo torna-se possível a criação de um sistema que garante a Capacidade de melhorar no principio de cada execução, característica que define o Anytime. Passa também a ser possível verificar a presença de Consistência na escolha dos blocos a gerar, conseguindo ainda confirmar qual a Qualidade Mensurável e Reconhecível das escolhas efetuadas pelo algoritmo. Pode assim dizer-se que este algoritmo serve como núcleo do Sistema Anytime desenvolvido. A Monotonicidade encontra-se também aqui presente pois os resultados obtidos pelo algoritmo estão diretamente dependentes da qualidade (ou número de casos) da amostra inicial presente no ficheiro XML. Para além disso, à medida que o tempo de execução aumenta o algoritmo passa a conseguir tomar decisões mais rapidamente, pois o número de novas regras a adicionar à tabela de regras vai decrescendo substancialmente com o tempo, conseguindo ainda tornar as

```
***** Count: 0 ****
2 | 2 | 2 | Grass
1 | 2 | 2 | Stone
3 | 2 | 2 | Snow
2 | 3 | 2 | Sand
3 | 1 | 2 | Snow
1 | 2 | 3 | Stone

***** Count: 16844 ****
2 | 2 | 2 | Grass
1 | 2 | 2 | Stone
3 | 2 | 2 | Snow
2 | 3 | 2 | Sand
3 | 1 | 2 | Snow
1 | 2 | 3 | Stone
2 | 1 | 2 | Grass

***** Count: 44796 ****
2 | 2 | 2 | Grass
1 | 2 | 2 | Stone
3 | 2 | 2 | Snow
2 | 3 | 2 | Sand
3 | 1 | 2 | Snow
1 | 2 | 3 | Stone
2 | 1 | 2 | Grass
1 | 1 | 2 | Stone

***** Count: 53616 ****
2 | 2 | 2 | Grass
1 | 2 | 2 | Stone
3 | 2 | 2 | Snow
2 | 3 | 2 | Sand
3 | 1 | 2 | Snow
1 | 2 | 3 | Stone
2 | 1 | 2 | Grass
1 | 1 | 2 | Stone
2 | 1 | 1 | Grass
```

```
***** Count: 54046 ****
2 | 2 | 2 | Grass
1 | 2 | 2 | Stone
3 | 2 | 2 | Snow
2 | 3 | 2 | Sand
3 | 1 | 2 | Snow
1 | 2 | 3 | Stone
2 | 1 | 2 | Grass
1 | 1 | 2 | Stone
2 | 1 | 1 | Grass
3 | 1 | 1 | Snow

***** Count: 60584 ****
2 | 2 | 2 | Grass
1 | 2 | 2 | Stone
3 | 2 | 2 | Snow
2 | 3 | 2 | Sand
3 | 1 | 2 | Snow
1 | 2 | 3 | Stone
2 | 1 | 2 | Grass
1 | 1 | 2 | Stone
2 | 1 | 1 | Grass
3 | 1 | 1 | Snow
1 | 1 | 1 | Stone

***** Count: 543209 ****
2 | 2 | 2 | Grass
1 | 2 | 2 | Stone
3 | 2 | 2 | Snow
2 | 3 | 2 | Sand
3 | 1 | 2 | Snow
1 | 2 | 3 | Stone
2 | 1 | 2 | Grass
1 | 1 | 2 | Stone
2 | 1 | 1 | Grass
3 | 1 | 1 | Snow
1 | 1 | 1 | Stone
2 | 2 | 1 | Grass
```

```
***** Count: 3035601 ****
2 | 2 | 2 | Grass
1 | 2 | 2 | Stone
3 | 2 | 2 | Snow
2 | 3 | 2 | Sand
3 | 1 | 2 | Snow
1 | 2 | 3 | Stone
2 | 1 | 2 | Grass
1 | 1 | 2 | Stone
2 | 1 | 1 | Grass
3 | 1 | 1 | Snow
1 | 1 | 1 | Stone
2 | 2 | 1 | Grass
1 | 2 | 1 | Stone
```

Figura 5.8: Evolução do número de regras ao longo do tempo.

decisões tomadas mais consistentes, pois as novas regras serão adicionadas com base em conhecimento previamente adquirido.

De referir, ainda, que a versão de .Net usada pelo Unity também não possui suporte nativo ao System.Data, biblioteca necessária devido ao uso da estrutura de dados Datatable, pelo que o ficheiro .DLL correspondente tem que ser importado manualmente a partir dos ficheiros presentes na pasta Mono do diretório onde se encontra o Unity.

5.3.3 Implementação do Algoritmo em Unity

A implementação do algoritmo em Unity foi feita substituindo o código básico que define os blocos de acordo com a altura por uma chamada ao script RuleSystem que representa o sistema de regras e cujo pseudo-código é dado pelos algoritmos 4 e 5 na secção 5.3.4.

Para possibilitar a sua utilização, no entanto, o sistema de regras tem obrigatoriamente que ser inicializado com o "mundo" (World), sendo para isso adicionada a função Start(),

utilizável por qualquer script que estenda o "MonoBehaviour" e que executa uma série de ações assim que o jogo começa, encontrando-se o pseudo-código correspondente a este comportamento presente no Algoritmo 3 na secção 5.3.4.

De modo a garantir que a oscilação dos valores do ambiente (Temperatura e Humidade) é feita de forma consistente, estes valores são apenas alterados após a geração de cada coluna de blocos de acordo com uma percentagem de oscilação definida no ficheiro XML utilizando um valor aleatório de zero a cem para determinar se os valores de ambiente sobem ou descem, tentando assim evitar que sejam gerados "Chunks" em que estes valores se apresentam constantes para todos os blocos neles contidos. Por exemplo, se este valor estiver definido no ficheiro XML como sendo 10 (dez por cento), caso o valor aleatoriamente gerado seja igual ou inferior a 5 o valor de ambiente irá descer enquanto que se este valor aleatório for superior ou igual a 95 o valor de ambiente irá subir. Note-se que, uma vez que se usam valores aleatórios para provocar oscilações nas variáveis de ambiente o processo de decisão não é estritamente determinístico (o algoritmo 6 na secção 5.3.4 apresenta o pseudo-código deste segmento do sistema).

De modo a demonstrar a forma como as regras afetam o tipo de terreno gerado, são dados dois exemplos de terrenos gerados de acordo com regras diferentes presentes no ficheiro XML nas figuras 5.9 a 5.12.

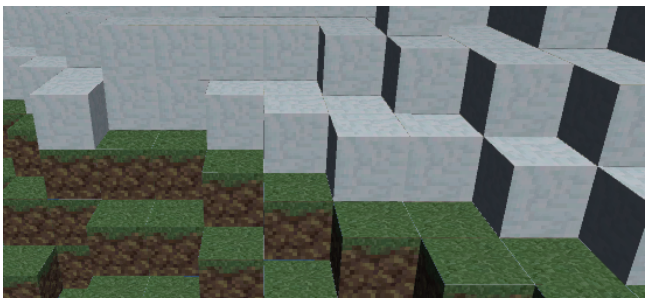

Figura 5.9: Terreno gerado de acordo com o conjunto de Regras da Figura 5.10.

```
<Rules>
  <Rule Height="Med"  Temperature="Med"  Humidity="Med" Block="Grass" />
  <Rule Height="Low"  Temperature="Med"  Humidity="Med" Block="Stone" />
  <Rule Height="High" Temperature="Med"  Humidity="Med" Block="Snow"  />
  <Rule Height="Med"  Temperature="High" Humidity="Med" Block="Sand"  />
  <Rule Height="High" Temperature="Low"  Humidity="Med" Block="Snow"  />
</Rules>
```

Figura 5.10: Regras usadas para gerar o terreno da Figura 5.9.

É desta forma possível comprovar a presença de variações no terreno de acordo com o conjunto de regras inicialmente passado à aplicação, sendo que regras iniciais diferentes resultam em terrenos com propriedades diferentes.

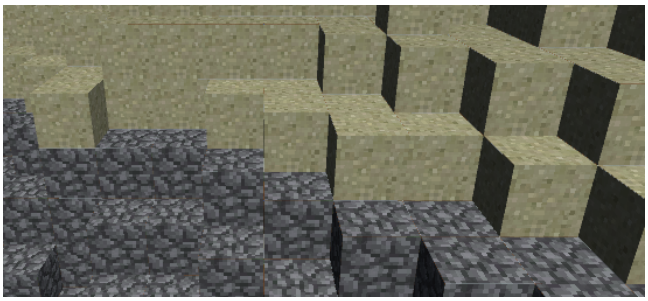

Figura 5.11: Terreno gerado de acordo com o conjunto de Regras da Figura 5.12.

```
<Rules>
  <Rule Height="Med"  Temperature="Med"  Humidity="Med" Block="Stone" />
  <Rule Height="Low"  Temperature="Med"  Humidity="Med" Block="Grass" />
  <Rule Height="High" Temperature="Med"  Humidity="Med" Block="Sand"  />
  <Rule Height="Med"  Temperature="High" Humidity="Med" Block="Snow"  />
</Rules>
```

Figura 5.12: Regras usadas para gerar o terreno da Figura 5.11.

5.3.4 Pseudo-código do Algoritmo

Apresenta-se agora o pseudo-código do algoritmo de decisão, pretendendo facilitar implementações futuras dando algum contexto. Note-se que o pseudo-código apresentado é aplicável única e exclusivamente ao caso em mãos, sendo necessário adaptar o pseudo-código presente neste anexo consoante a situação.

Algoritmo 3 Pseudo-código do gerador

Require: $Chunk$ iterável composto por $Block$ com coordenadas x, y e z e um $BlockType$ que define que tipo de bloco é que este representa no espaço de jogo, bem como a existência de um $File f$ e das variáveis $vTemp$ e $vHumid$

Ensure: Cada bloco no $Chunk$ recebido como input passe a estar representado no mapa e as variáveis $vTemp$ e $vHumid$ são alteradas a cada iteração

$File\ f$
$DecisionSystem\ d \leftarrow initTable(f)$
$vTemp \leftarrow startingTemp$ from d
$vHumid \leftarrow startingHumidity$ from d

function GETTERRAIN($Chunk$)

for each $Block\ b$ in $Chunk\ c$ **do**
$BlockType$ in $b \leftarrow checkRules(y$ in $b, vTemp, vHumid)$ from d
$vTemp \leftarrow oscillate(vTemp)$ from d
$vHumid \leftarrow oscillate(vHumid)$ from d
end for

end function

Algoritmo 4 Inicialização do sistema

Require: $startingTemp$, $startingHumidity$, $variationChance$, $lowerChance$, $upperChance$ e $ruleTable$ instanciados, pressupondo-se a existência de um $Filef$ e onde NaN é qualquer valor não numérico

Ensure: Inicializa e preenche a Tabela de Regras $ruleTable$ de acordo com os dados presentes em f e atribui os valores às variáveis $startingTemp$, $startingHumidity$, $lowerChance$, $upperChance$

function DECISIONSYSTEM(f)
read $Filef$
get $startingTemp$ from f
if $startingTemp = NaN$ **then**
$startingTemp \leftarrow randomInt(0, 100)$
end if

get $startingHumidity$ from f
if $startingHumidity = NaN$ **then**
$startingHumidity \leftarrow randomInt(0, 100)$
end if

get $variationChance$ from f
if $variationChance \leq 0$ **and** $variationChance \geq 100$ **then**
$variationChance \leftarrow 50$
end if
$upperChance \leftarrow 100 - variationChance/2$
$lowerChance \leftarrow variationChance/2$

set $ruleTable$ to $Height$, $Temp$, $Humidity$ **and** $Block$
get rules from f into $ruleTable$

end function

Algoritmo 5 Verificação e Inferência de Regras

Require: $ruleTable$ contém as colunas $Height$, $Temp$, $Humidity$ e $Block$ e existe um $defaultBlock$ instanciado

Ensure: Retorna o tipo de bloco $result$ criando uma nova regra na tabela de regras $ruleTable$ se não existir uma regra correspondente aos valores de $height$, $temp$ e $humid$

function CHECKRULES($height$,$temp$,$humid$)

$result \leftarrow firstBlock$ **from** $ruleTable$ **where**
$Height = height$ **and** $Temp = temp$ **and** $Humidity = humid$

if $result$ = $NULL$ **then**
$result \leftarrow firstBlock$ **from** $ruleTable$ **where**
($Height = height$ **and** $Temp = temp$) **or**
($Height = height$ **and** $Humidity = humid$) **or**
($Temp = temp$ **and** $Humidity = humid$) **or**
group by count
order by descending
end if

if $result$ = $NULL$ **then**
$result \leftarrow defaultBlock$
end if

add $rule(height, temp, humid, result)$ to $ruleTable$

return $result$

end function

Algoritmo 6 Oscilação de valores

Require: val seja um valor numérico, $lowerChance$ e $upperChance$

Ensure: val varia de acordo com os valores de $lowerChance$ e $upperChance$, aumentando se superior a $upperChance$, diminuindo se inferior a $lowerChance$ e mantendo-se igual caso contrário

function OSCILLATE(val)

$r \leftarrow random(0, 100)$

if $r \leq lowerChance$ **and** $val > 0$ **then**
$val \leftarrow val - 1$
end if

if $r \geq lowerChance$ **and** $val < 100$ **then**
$val \leftarrow val + 1$
end if

return val

end function

5.3.5 Sumário

Neste capítulo foram detalhados cada um dos elementos que compõem a aplicação desenvolvida para o projeto. O sistema de geração de terreno aqui apresentado usa blocos do tipo de terreno a gerar para construir um mapa de acordo com um conjunto de regras predefinido num ficheiro XML que permite a alteração das variáveis de inicio do programa sem que haja a necessidade de alterar o código fonte da aplicação para obter terrenos de acordo com dados (ou amostras) iniciais diferentes, garantindo à aplicação flexibilidade no que toca à geração de novos cenários de acordo com o conjunto de regras inicialmente lido.

Inicialmente pretendia fazer-se uso de Prolog para a criação do sistema de regras, algo tornado simples por esta linguagem. No entanto, tendo surgido problemas relacionados com a eficiência e implementação do código da interface entre a versão da framework .Net suportada pelo Unity e o Prolog, esta ideia foi descartada a favor de uma implementação de um sistema de regras escrito exclusivamente em C# que evita estes problemas por tornar a implementação de uma interface entre o Sistema de Regras e o Gerador de Terreno desnecessária.

Ao ler as regras presentes na amostra inicial, o algoritmo de geração obtém um conjunto de variáveis de ambiente (Temperatura e Humidade) usadas na criação de novos blocos de acordo com o conjunto de regras presente na amostra inicial. Cada uma destas regras encontra-se armazenada numa tabela que é usada como forma de comparar os valores dos

dados numa dada altura da execução com o conhecimento presente no sistema de regras, sendo cada bloco gerado pelo algoritmo um resultado direto desta análise. Casos em que não sejam encontradas regras às quais correspondam os valores em questão, são geradas e adicionadas à base de conhecimento novas regras para futura referência. Isto é conseguido através de um sistema de inferência que analisa todo o conhecimento presente na tabela de regras, criando uma aproximação a uma nova regra de acordo com os dados segundo os quais se pretende efetuar a geração de um novo bloco.

Para garantir a eficiência da aplicação este sistema de análise de casos é executado apenas quando existe a necessidade de gerar novos "Chunks" de blocos, monitorizando-se para o efeito de forma constante a posição do objeto de jogo correspondente ao jogador, sendo novos blocos criados apenas quando este se aproxima de zonas vazias no mapa, tornando esta característica do sistema na principal componente do sistema Anytime. São também removidos "Chunks" que se encontrem a uma distância demasiado grande do jogador, assegurando que os "Chunks" em memória não são numerosos a ponto de interferir com a eficiência da aplicação. Faça-se notar que cada "Chunk" criado é guardado num diretório específico para permitir a reutilização de "Chunks" previamente gerados sem que haja a necessidade de voltar a correr o algoritmo de geração, recorrendo-se para o efeito a serialização.

Foi ainda criado um gerador de terreno simplificado, possuindo uma complexidade muito menor do que o original, para efeitos de teste de eficiência, não possuindo este controlo sobre "Chunks" e Blocos nem a capacidade de carregar blocos a partir de ficheiros usando serialização.

Há que notar que graças ao processo de aprendizagem descrito, a geração do terreno vai-se tornando mais rápida pois a necessidade de adicionar novas regras à base de conhecimento decresce com cada novo elemento na tabela de regras. Torna-se, deste modo, possível implementar um algoritmo de decisão baseado em regras sobre um sistema Anytime não só inovador (pela falta de presença deste tipo de algoritmos a nível académico), como também possuidor de um nível de eficiência crescente.

Capítulo 6

Resultados

Ao longo do desenvolvimento do projeto efetuaram-se múltiplos testes a cada um dos elementos que compõem a aplicação final. Neste capítulo apresentam-se os resultados obtidos a partir destes testes, procurando explicar tais resultados o mais detalhadamente possível, assinalando os problemas que surgiram ao longo do desenvolvimento da aplicação que conduziram à necessidade de recorrer a métodos alternativos que contribuíssem para que o desenvolvimento do projeto fosse viável.

Salienta-se que os resultados que se apresentam, se obtiveram numa máquina com as seguintes especificações:

- Sistema Operativo: Windows 10 Education 64-bit
- CPU: Intel Core i7 3630QM, 2.40GHz
- RAM: 32,0GB
- Drive: Intel SSD 520 Series

A avaliação de cada algoritmo foi feita de acordo com o tempo que cada aplicação demorou a executar uma determinada tarefa. Apresentam-se sempre que possível os valores do tempo, de forma detalhada, exceto em casos em que a diferença resultante da comparação de tempos entre elementos é evidente.

6.1 Geradores de Terreno

Durante o desenvolvimento da aplicação original ocorreram problemas na implementação do algoritmo de geração da altura por Noise, que conduziram à procura de uma alternativa que fosse mais simples de implementar e de aplicar a outros casos.

Este fator permitiu a criação de uma aplicação de geração de terreno mais simples que faz apenas uso das funcionalidades do Unity para gerar novos blocos de acordo com a posição do jogador.

A versão inicial desta aplicação foi desenhada de modo a gerar apenas uma única camada de blocos. A implementação do algoritmo de "Noise" na geração de relevo apresentou falhas, permitindo a existência de buracos no espaço gerado através dos quais o jogador poderia passar, provocando depois a sua queda por tempo indeterminado. Em casos extremos, ou seja, casos em que a variação entre a altura dos blocos é significativa, estas falhas são facilmente visíveis, como se pode observar na figura 6.1 pelas zonas a cinza.

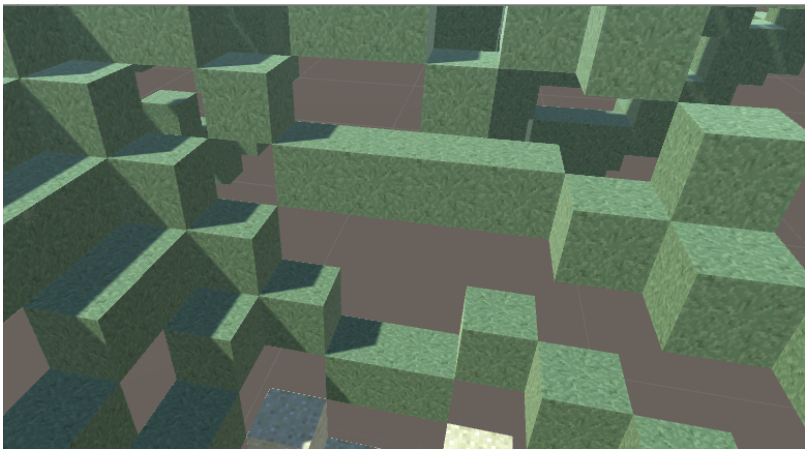

Figura 6.1: Exemplo de um caso extremo.

Os testes aos geradores de terreno nesta fase foram feitos de modo a verificar a sua eficiência no que toca à geração de terrenos de larga escala (define-se larga escala como um espaço com pelo menos cento e cinquenta blocos de comprimento e largura).

Os resultados destes testes revelaram que enquanto o Gerador de terreno implementado inicialmente conseguia gerar um terreno de larga escala, com profundidade considerável (aproximadamente trinta e dois blocos), em poucos segundos, o gerador simplificado demorava muito mais tempo a gerar o mesmo tipo de terreno com uma profundidade de um único bloco.

Durante o tempo em que a aplicação esteve a correr, o gerador simplificado apresentou problemas de FPS (Frames per Second), sendo que a velocidade do jogo após a geração do terreno caía subitamente logo que esta ação terminava. Em contrapartida, o gerador original conseguia manter um nível de FPS estável, não se verificando grandes oscilações durante o "runtime" do jogo.

Enquanto o gerador original funcionava de modo a organizar blocos em grupos, removendo faces e segmentos cuja presença no mapa fosse desnecessária (pois o jogador não deveria ver ou interagir com esses elementos), o gerador simplificado não o faz. Isto permite à aplicação de exemplo original manter, não só, eficiência na geração, como também durante o "runtime".

Esta situação verifica-se principalmente porque o sistema simplificado se limita a gerar conjuntos de blocos, usando um grupo de "Prefabs" aos quais acede, sem controlar quais

os lados ou elementos que devem ser renderizados no mapa, algo que resulta no facto de todas as faces de todos os blocos serem geradas, independentemente do número de camadas presente num grupo de blocos.

Esta falta de controlo provoca um aumento significativo na memória usada em função da quantidade de blocos usados, uma vez que cada bloco gerado é tratado de forma individual.

O gerador original consegue fundir cada conjunto de faces laterais de um grupo de blocos numa só face, fazendo com que cada "Chunk" seja composto apenas pelo conjunto exterior de faces do sólido, independentemente da sua dimensão, o que reduz substancialmente a quantidade de recursos usados.

Verificou-se que, no caso do gerador secundário, quantos mais blocos forem gerados, mais memória será necessária. Comparativamente o gerador original, trata os blocos de forma individualizada, ao agrupar os blocos em "Chunks" e ao remover as faces invisíveis ao jogador no processo de geração. O gerador original consegue gerar blocos em números superiores ao gerador secundário, enquanto mantém um nível de eficiência aceitável.

Comparando a eficiência de cada um dos geradores, decidiu-se utilizar o gerador original, pois apesar de possuir um número bastante elevado de "peças" interligadas, a eficiência na execução do exemplo revelou-se um fator de extrema importância, o que o coloca acima do gerador secundário, uma vez que, apesar de este ser mais fácil de usar, alterar e implementar, é extremamente ineficiente quando está em causa a gestão de recursos, sendo por isso preferível recorrer ao gerador original.

6.2 Prolog

A implementação do algoritmo de Prolog inicialmente proposto previa uma interface usando o tuProlog de modo a permitir que as regras criadas no ficheiro de Prolog pudessem ser utilizadas num ambiente .Net.

Os factos eram adicionados através de chamadas diretas ao Prolog concretizadas a partir da interface, permitindo que todas as regras pudessem ser adicionadas a partir do C# no inicio do algoritmo, ou durante o runtime da aplicação, garantindo ao utilizador um controlo elevado sobre o conteúdo do algoritmo e, consequentemente sobre os resultados gerados pela aplicação.

O código Prolog inicial continha, na altura do desenvolvimento desta secção do projeto, apenas regras que permitiam a geração de blocos de acordo com a temperatura de um bloco, valor este que variava a cada bloco gerado.

Foram efetuados testes usando o VisualStudio com vista a facilitar a integração de uma interface já funcional no Unity. Os métodos testados foram criados com o propósito de fazer "Assert" e "Retract" à base de factos, e chamadas às regras que permitiam a oscilação de valores e geração de blocos. Como o retorno destes testes se enquadrava nos resultados

esperados no âmbito dos dados anteriormente obtidos, testou-se a implementação do algoritmo em Unity.

Não foi possível terminar, em tempo aceitável, os testes em Unity, uma vez que apesar do comportamento da aplicação se apresentar de acordo com o esperado, o algoritmo era de tal forma lento a responder que foi impossível conseguir ver um único "Chunk" a ser gerado.

Uma análise à velocidade da interface revelou que, ao repetir o teste inicial, que passava por efetuar o cálculo correspondente ao vigésimo número da sequência de Fibonacci para verificar a velocidade do Prolog, a interface demorou 0.387 segundos, podendo assim afirmar-se de que a interface desenvolvida apresentava problemas graves e significativos de eficiência face ao Prolog sem interface.

Ao investigar o porquê desta falta de eficiência, foi encontrada informação relativa ao funcionamento da "framework" tuProlog, um conjunto de bibliotecas que se encontra basicamente dependente da ferramenta IKVM. Esta ferramenta é fundamentalmente constituída por três elementos [4]:

- Uma máquina virtual Java implementada em C#;
- Uma implementação em .Net das bibliotecas de Java;
- Ferramentas que permitem a interoperabilidade entre Java e C#;

Com isto podemos dizer que as ferramentas para o estabelecimento da ligação com a linguagem Prolog são geradas e desenvolvidas utilizando IKVM, o que resulta num conjunto de ferramentas Java em .Net . Apesar de ser mais fácil de desenvolver ambas as versões da ferramenta em conjunto, a versão .Net não é particularmente eficiente, sendo 15% mais lenta do que a versão Java [16].

Tendo estes valores em conta, procurou-se uma alternativa, mais eficiente, que continuasse a permitir o uso do código Prolog anteriormente desenvolvido, tendo-se escolhido para o efeito o SWI-pl.

Esta escolha resulta de uma análise comparativa da velocidade entre esta "framework" e a providenciada pelo tuProlog. Procedeu-se à repetição dos testes para verificar qual o tempo que cada uma das interfaces demorava a executar um pedido feito ao Prolog (utilizando novamente o cálculo $F(20)$).

Da comparação dos resultados obtidos pela interface tuProlog com a interface SWI-pl, concluiu-se que, enquanto a primeira interface demora 0.387 segundos mencionados, a segunda demora apenas 0.074 segundos, o que representa uma diferença considerável.

Tendo em conta que a sintaxe não difere significativamente da inicialmente utilizada, procedeu-se à transferência de todas as funcionalidades originais para a nova interface, e por aparentemente não apresentar qualquer tipo de erros, procedeu-se à sua implementação no ambiente Unity.

Mais uma vez os testes em Unity não deram um resultado válido, ou seja, o sistema de exemplo não gerou qualquer tipo de bloco ou "Chunk". Estes resultados deveram-se não a problemas de eficiência, mas porque a biblioteca usada para estabelecer a interface não é compatível com a distribuição da "framework" de .Net utilizada pelo Unity. Ao correr o código não ocorre a geração de qualquer tipo de bloco, apresentando a consola de Debug do Unity uma NotImplementedException().

Uma investigação mais apurada possibilitou a determinação da origem desta exceção. O erro apresenta-se como uma exceção lançada pela "framework" .Net, usada tipicamente para definir que a funcionalidade que está a tentar ser usada ou acedida ainda se encontra por implementar na versão de .Net corrente.

Esta situação verifica-se porque o Unity não usa a versão da framework .Net mais recente, mas sim uma versão da framework desenvolvida especificamente para o compilador de C# Mono, que neste caso, engloba apenas a versão 2.0 do .Net, com uma quantidade limitada das funcionalidades presentes na versão 3.5 da "framework" distribuída pela Microsoft.

Como resultado desta exceção, são passados dados inválidos ao script de geração de blocos, que provoca um efeito dominó, fazendo com que qualquer tentativa de gerar um bloco cause uma outra exceção, impossibilitando deste modo, que o código dedicado a esta tarefa funcione corretamente.

Sendo este problema incontornável, é invalidada a hipótese de apresentar uma solução usando a interface com o C# providenciada pelo SWI-pl.

6.3 Algoritmo de Decisão e Inferência em C#

Após a análise dos resultados obtidos, decidiu-se abandonar o Prolog definitivamente, pelo que nos focámos no desenvolvimento de um algoritmo capaz de proceder a simulações próximas das conseguidas pelo Prolog, facilitando deste modo, a sua integração no Unity.

A utilização de um ficheiro XML para definir factos e regras (como o da figura 6.2) permitiu que os dados recebidos pelo algoritmo fossem facilmente alterados, tornando-o num algoritmo flexível ao nível da usabilidade no que respeita à adição e remoção de regras na amostra inicial.

```
<Rules>
    <Rule Height="Med"  Temperature="Med"  Humidity="Med"  Block="Dirt"  />
    <Rule Height="Low"  Temperature="Med"  Humidity="Med"  Block="Stone" />
    <Rule Height="High" Temperature="Med"  Humidity="Med"  Block="Snow"  />
    <Rule Height="Med"  Temperature="High" Humidity="Med"  Block="Sand"  />
    <Rule Height="Med"  Temperature="Low"  Humidity="Med"  Block="Snow"  />
    <Rule Height="Med"  Temperature="Med"  Humidity="High" Block="Mud"   />
</Rules>
```

Figura 6.2: Tabela de Regras definida em XML.

A utilização da estrutura de dados "DataTable" permitiu a implementação da biblioteca LINQ, que tornou todo o algoritmo mais fácil de manter e desenvolver através da

substituição de ciclos sobre código por queries simples feitas à tabela, sendo que o LINQ é, para este tipo de situações em que as queries são simples, normalmente mais fácil de manter do que um conjunto de ciclos desenhados para iterar sobre cada linha presente na tabela.

```
res = from DataRow r in ruleTable.Rows
      where ((Values)r["Height"]   == height   && (Values)r["Temperature"] == temperature ) ||
            ((Values)r["Height"]   == height   && (Values)r["Humidity"]    == humidity    ) ||
            ((Values)r["Humidity"] == humidity && (Values)r["Temperature"] == temperature )
      select r["Block"];

res = from r in res group r by r into grp orderby grp.Count() descending select grp.Key;

block = (String)res.FirstOrDefault();
```

Figura 6.3: Exemplo de uma query feita em LINQ.

Nos novos resultados foi possível confirmar que o algoritmo de decisão escolhe qual o tipo de bloco a retornar de forma mais "lógica" do que anteriormente, sendo a resposta obtida, a esperada pelo utilizador após uma rápida verificação dos dados presentes na tabela.

Foi possível confirmar também que, para o mesmo conjunto de condições, o algoritmo devolve sempre o mesmo tipo de bloco. Verificando-se ainda um aumento da velocidade (visível a olho nú durante a execução da aplicação) da resposta após a adição de uma regra previamente inexistente à tabela de regras, não sendo por isso necessário efetuar a segunda query novamente para obter a resposta.

Ao analisar os resultados obtidos nos testes feitos ao algoritmo, foi possível confirmar que este consegue garantir a presença de muitas das características que compõem um bom algoritmo integrável em Anytime, pois os dados retornados pelo algoritmo permitem verificar que este possui elementos que garantem a Certeza, Precisão e Especificidade dos resultados obtidos.

O facto de ser um algoritmo que consegue aprender por si só permite, no decorrer de cada execução, a adição de múltiplas novas regras à tabela de regras, sendo que a quantidade de regras aprendidas vai diminuindo até que todos os casos possíveis estejam armazenados na tabela. Aqui garantimos a Capacidade de Melhorar que caracteriza os Algoritmos Anytime.

Para além disso, os resultados variam largamente de acordo com a amostra inicial lida a partir do ficheiro XML. Quantas mais regras estiverem presentes nesta amostra, mais precisos serão os resultados devolvidos pelo sistema. Tendo em conta que o sistema já possui a capacidade de melhorar os resultados retornados, pode garantir-se a Monotonicidade do programa.

Uma análise à tabela de dados e aos resultados obtidos permitiu confirmar a qualidade das respostas pois ao correr o algoritmo, os blocos retornados eram exatamente aqueles que eram esperados, tendo em conta o processo de decisão e inferência executado

pelo algoritmo, obtendo-se sempre o mesmo resultado para o mesmo conjunto de dados, comprovando-se assim a presença da Consistência nas respostas dadas, bem como da Qualidade Mensurável.

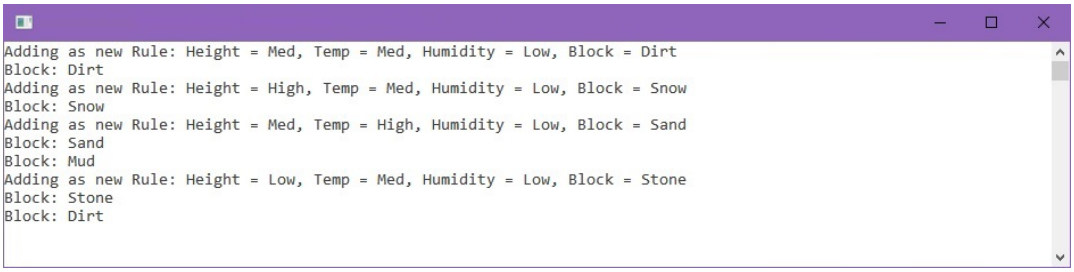

Figura 6.4: Resultados dados pelo Algoritmo em Visual Studio.

6.4 Implementação do Algoritmo em Unity

No decorrer do teste inicial, verificou-se a existência de um erro nos atributos que contêm os intervalos usados para a conversão de valores numéricos em Enumerados. Este bug afetava a atribuição dos Enumerados ao atributo, isto porque lhes era sempre atribuído o valor correspondente ao maior enumerado possível (High ou Max).

Para verificar se a oscilação de valores estava a ser devidamente efetuada pelo algoritmo, definiu-se que a probabilidade de ocorrerem alterações nas variáveis de ambiente seria 100%. Desta vez, o terreno foi de facto gerado de acordo com o esperado, como é possível verificar na figura 6.5. Era agora composto por blocos de diferentes tipos, dispostos ao longo do mapa de uma forma extremamente aleatória. Foi, no entanto, possível confirmar a geração do tipo de bloco correto de acordo com os valores guardados no sistema de regras através do debug dos resultados obtidos pela query LINQ feita à tabela de regras.

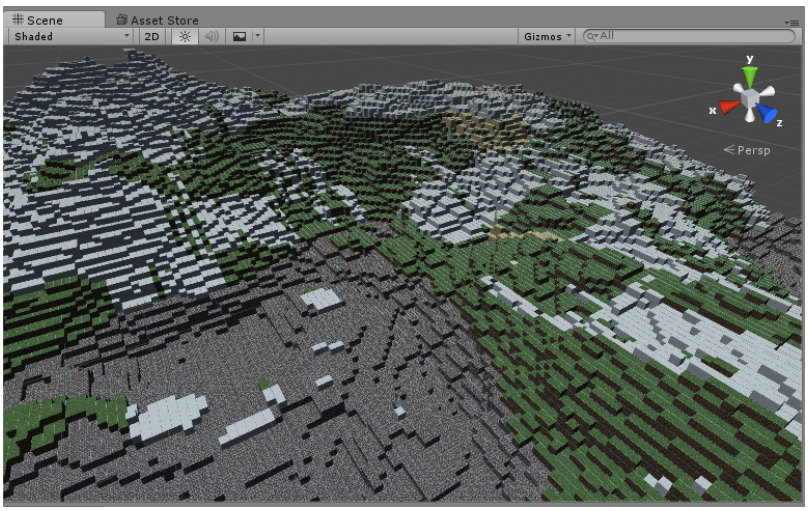

Figura 6.5: Exemplo de um terreno gerado pelo algoritmo.

É importante notar que, durante os testes à geração de terreno de acordo com o movimento, foi possível verificar uma ligeira diminuição no framerate do jogo no momento da geração dos novos "Chunks", constatando-se que logo após o processo de geração, o jogo voltou a correr à velocidade normal.

Não foi implementada uma "seed" para garantir a atribuição de alturas aleatórias ao terreno (a altura de cada bloco no mapa gerado será sempre o mesmo apesar de os blocos criados serem de diferentes tipos) para garantir que o algoritmo era capaz de produzir os resultados pretendidos, usando sempre o mesmo tipo de mapa como ponto de partida.

Foi também possível verificar que a diminuição no framerate ocorria com mais frequência nas ocasiões em que era adicionada uma nova regra à tabela, dando-se esta queda principalmente pela adição de novos elementos à base de conhecimento requerer cálculos mais complexos do que um simples acesso à tabela de dados.

A irregularidade presente no terreno gerado deve-se unicamente à constante variação dos valores de ambiente. Ao atribuir o valor de 10% à probabilidade de ocorrerem oscilações nas variáveis de ambiente, o resultado é um terreno muito mais uniforme, visível na figura 6.6. Este valor encontra-se definido no ficheiro de XML que contém os valores usados pelo gerador para a criação do mundo e é alterável por qualquer utilizador.

De modo a permitir a geração com um nível superior de irregularidade nestas variáveis foram alteradas as clausulas que lhes atribuem um valor numérico a partir do ficheiro XML. Após a alteração, se o valor recebido não for um inteiro, estas variáveis irão assumir um valor aleatório entre zero e cem.

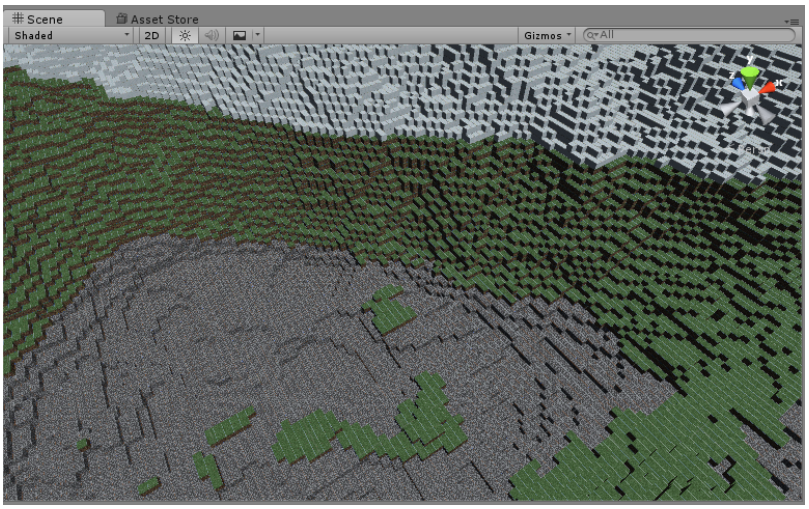

Figura 6.6: Exemplo de outro terreno gerado com o Algoritmo.

Por fim, resta referir que em certas ocasiões, o terreno gerado é predominantemente composto por um único tipo de bloco. Isto deve-se principalmente à dimensão reduzida da amostra inicial utilizada (que contém apenas cinco regras básicas). Dependendo da forma como os valores das variáveis de ambiente oscilam, o sistema começa a popular a

tabela de regras com casos em que o valor a retornar passa a ser um único bloco de um tipo específico, algo que pode ser corrigido substituindo o sistema de oscilação dos valores de ambiente corrente por uma fórmula que faça com que estes valores oscilem de forma mais natural.

6.5 Sumário

Uma análise comparativa entre a eficiência do gerador original e do secundário revelou que, ao gerar um número elevado de blocos, esta aplicação secundária começava a sofrer de graves problemas de Framerate, algo que não ocorria no gerador original, levando a que o gerador de terreno simplificado fosse descartado devido à ineficiência do código face à quantidade de elementos que o compunham. Note-se que, para garantir a consistência dos resultados, não foi aplicada a geração de acordo com regras nem foram utilizados valores aleatórios no momento da geração, sendo que o resultado final de ambas as gerações seria obrigatoriamente o mesmo.

Tendo sido definido sobre qual dos dois geradores seria implementado o sistema de decisão foram efetuados testes ao Prolog utilizando-se, para o efeito, código em C# cujo objetivo principal seria servir de interface entre o gerador em Unity e a base de regras em Prolog. Estes testes permitiram fazer uma análise ao código que serviria como interface entre a base de conhecimento e o gerador de regras, sendo que a primeira interface criada para o efeito foi descartada devido a graves problemas de eficiência relacionados com as bibliotecas necessárias ao estabelecimento da ligação entre as duas linguagens.

Procurando encontrar-se uma solução para este problema recorreu-se a um conjunto de bibliotecas alternativas cujo funcionamento interno era independente de muitas das ferramentas usadas pelas bibliotecas inicialmente selecionadas. Os testes à nova interface demonstraram que esta não sofria dos mesmos problemas que a original, podendo assim proceder-se à sua integração no sistema de geração de terreno. Esta integração veio mais tarde a provar-se impossível devido a disparidades entre as versões da framework .Net usada pelas bibliotecas e pelo Unity, tendo sido por isso necessário recorrer a outros métodos para o desenvolvimento da base de conhecimento que não o Prolog.

Foi com o objetivo de simular o comportamento de um sistema de regras em Prolog que se desenvolveu o algoritmo de decisão em C# correntemente utilizado pela aplicação. Os testes a este sistema trataram de verificar variações na geração de terreno bem como a consistência das respostas dadas pelo algoritmo dado um determinado conjunto de dados, pois o resultado tem que ser sempre igual para o mesmo conjunto de dados e regras (algo comprovado após os testes ao sistema).

Finalmente, deu-se a integração do sistema de regras no gerador, tendo sido adicionados elementos relativos à oscilação de valores que visam garantir a presença de elementos de aleatoriedade no mapa. Neste caso não foi implementada uma “seed” para causar variações

no terreno gerado com cada execução pois, para efeito de teste, existe a necessidade de comparar detalhadamente cada iteração do terreno gerado. Neste caso foram verificadas flutuações no framerate visíveis ao olho humano quando se dá a inserção de um novo elemento na base de conhecimento, algo que ocorre devido à complexidade dos cálculos envolvidos.

Capítulo 7

Conclusão

No projeto em que assenta este trabalho foi implementado um algoritmo de regras que funcione em Anytime fazendo uso de um Sistema baseado em Regras para a tomada de decisões. A falta de trabalhos académicos relativos aos temas relacionados com Anytime ou Sistemas Baseados em Regras aplicados a videojogos faz-nos acreditar que a investigação nesta área é limitada ou feita principalmente a um nível comercial/empresarial, uma vez que raras são as referências das empresas de desenvolvimento de videojogos relativas aos sistemas de Inteligência Artificial usados nos produtos que distribuem.

O sistema resultante permite a geração de terreno através de um ficheiro XML onde são definidos um conjunto de Factos e Regras. O terreno resultante da geração tende a variar de acordo com o número de regras presentes, tal como a probabilidade de ocorrer uma oscilação nos valores das variáveis de ambiente presentes no mundo (temperatura e humidade). Todos os valores necessários ao funcionamento do algoritmo podem ser facilmente editados no ficheiro XML, sendo este posteriormente lido pelo algoritmo e utilizado para construir a tabela de regras usada na seleção do bloco a gerar. Isto permite uma fácil edição dos fatores que condicionam a geração do terreno, permitindo que um utilizador do algoritmo não tenha que fazer alterações ao código para obter um resultado diferente. A Monotonicidade está aqui presente como um aspeto que caracteriza o Anytime neste ponto pois a qualidade e o tipo do terreno gerado variam principalmente de acordo com a qualidade da amostra de regras fornecida no início de cada execução. Há ainda que referir que o algoritmo está desenhado de modo a aprender a lidar com novos casos à medida que o tempo de execução aumenta. No principio da execução o algoritmo irá preencher a sua base de regras com cada caso que ainda não tenha sido encontrado, tornando-se assim cada vez mais rápido à medida que o tempo passa. Para além disso, regras adicionadas à base de regras em fases mais tardias da execução são geradas tendo em conta todas as que já lá estão presentes, resultando em regras de qualidade superior às iniciais.

Este sistema consegue facilmente selecionar o bloco pretendido, aprendendo a gerar blocos em situações que não constam na tabela de regras, usando o conhecimento previa-

mente adquirido como base para a criação de novas regras. Algo que garante a presença de uma capacidade de aprendizagem tipicamente existente nos Algoritmos Anytime. O facto de novo terreno ser gerado apenas quando o jogador se aproxima de uma zona sem terreno demonstra tanto a Interruptibilidade do algoritmo, bem como a Capacidade de recuperação de Interrupções do mesmo, pois o algoritmo não é executado de forma constante, sendo corrido apenas quando necessário.

Para efeitos de teste não foi implementada a geração aleatória de terreno (o mapa gerado será sempre o mesmo, com blocos de diferentes tipos) de modo a garantir que o algoritmo é capaz de produzir os resultados pretendidos, usando sempre o mesmo tipo de mapa como ponto de partida, apresentando deste modo a Consistência dos resultados obtidos e garantindo uma boa Qualidade Mensurável, algo que caracteriza também um algoritmo Anytime.

Este algoritmo apresenta algumas complicações, como por exemplo o facto de estar limitado à linguagem C# devido ao uso do LINQ. Em contrapartida, o uso de LINQ não só facilita a leitura e desenvolvimento do código, como também torna a sua manutenção mais fácil, pois permite evitar a construção de ciclos para iterar sobre a tabela de regras através do uso de queries simples similares às usadas em SQL sendo, no entanto, menos eficiente neste caso do que a aplicação de conjuntos de ciclos.

Seguem-se as limitações sofridas pelo sistema, bem como potenciais pontos de partida para trabalho futuro na área.

7.1 Limitações

- O Algoritmo funciona apenas em C# devido ao uso da biblioteca LINQ;
- A implementação corrente do algoritmo está desenhada para o caso específico em que existem apenas três atributos;
- Quanto maior o número de regras definidas melhor será o resultado final, sendo que, com um número reduzido de regras, o terreno gerado pode vir a povoar a tabela de regras sempre com o mesmo tipo de bloco;
- O parser de XML está desenhado de modo a aceitar apenas o tipo de estrutura presentemente definido.

7.2 Trabalho Futuro

- Implementar um sistema de regras usando Prolog num Motor de Jogo que não o Unity;

- Implementar um sistema de regras usando Prolog sobre o Unity após este receber um update à versão de .Net usada;
- Generalizar o algoritmo de modo a facilitar a sua implementação em diferentes situações;
- Permitir a implementação do algoritmo noutras linguagens que não o C#;

O planeamento inicial para o projeto não foi cumprido na íntegra. A descoberta de um tutorial que cobria a maioria do desenvolvimento do gerador de terreno permitiu saltar vários dos passos delineados no principio do projeto. Graças a este avanço foi possível atribuir mais tempo ao desenvolvimento da componente baseada em regras, muito do qual foi perdido quando ocorreram as falhas na implementação das interfaces com o Prolog. Não foi, por isso possível, criar uma aplicação que fosse 100% de acordo com as métricas inicialmente delineadas.

O algoritmo apresentado consegue fazer uso de um conjunto de regras simples para gerar novas regras em situações sobre as quais não existe conhecimento prévio, aprendendo assim a lidar com novas situações, garantindo sempre a consistência das decisões tomadas e apresentando um impacto negligenciável no desempenho da aplicação em si, podendo assim afirmar-se que foram cumpridos todos os requisitos principais propostos. É ainda possível a geração procedimental de terreno, sendo esta aplicável a outros casos sem que haja a necessidade de efetuar grandes alterações à estrutura do código apresentada, podendo dizer-se com base na investigação feita anteriormente que o algoritmo apresentado é um elemento inovador numa área onde a informação sobre o tópico em mãos é escassa, conseguindo estabelecer uma base para pesquisa futura na área do Anytime aplicado a jogos.

Apêndice A

Modelo UML da aplicação parte 1.

Figura A.1: Modelo completo - http://bit.ly/2cvRd8F

Apêndice B

Modelo UML da aplicação parte 2.

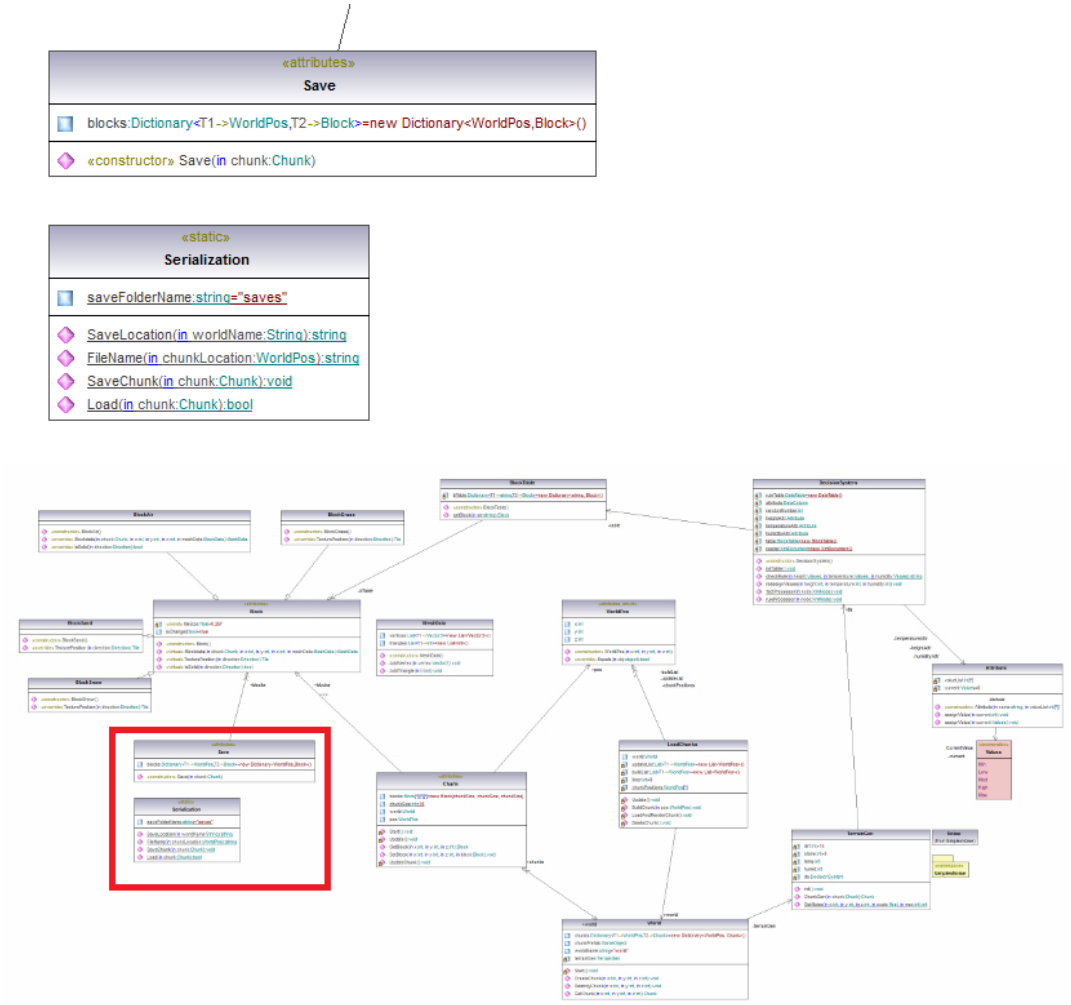

Figura B.1: Modelo completo - http://bit.ly/2cvRd8F

Apêndice C

Modelo UML da aplicação parte 3.

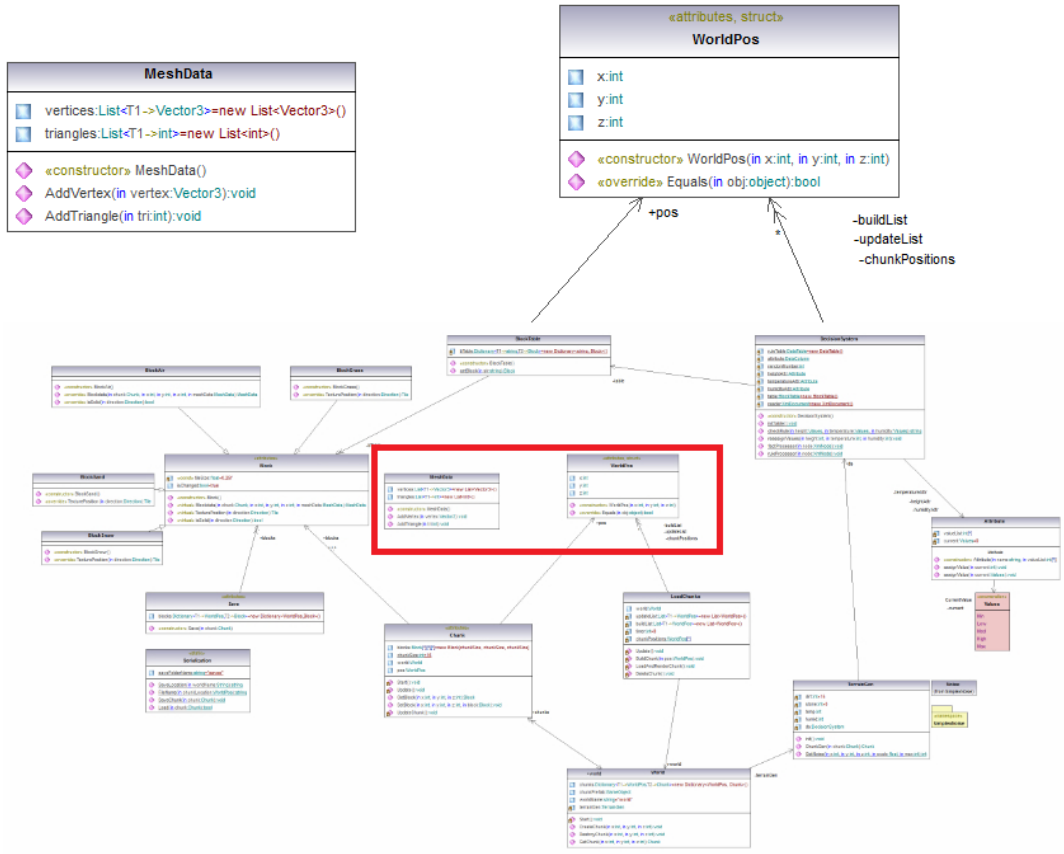

Figura C.1: Modelo completo - http://bit.ly/2cvRd8F

Apêndice D

Modelo UML da aplicação parte 4.

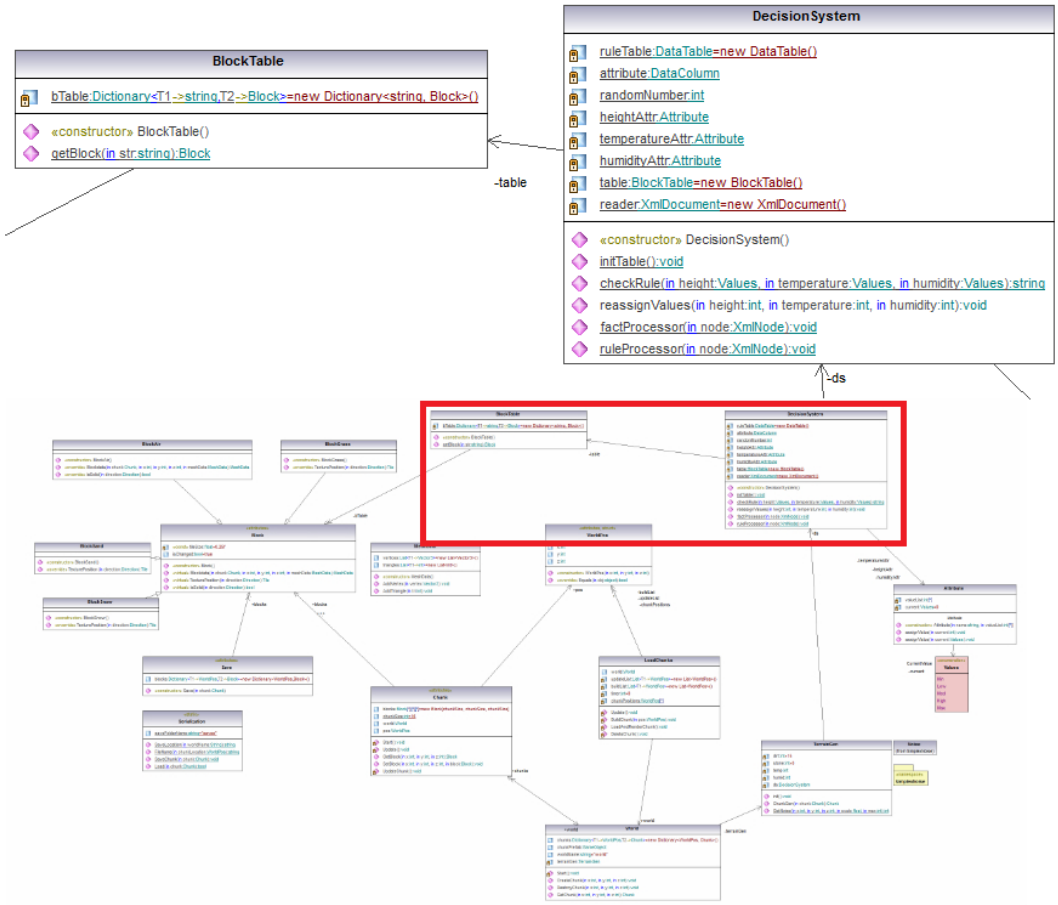

Figura D.1: Modelo completo - http://bit.ly/2cvRd8F

Apêndice E

Modelo UML da aplicação parte 5.

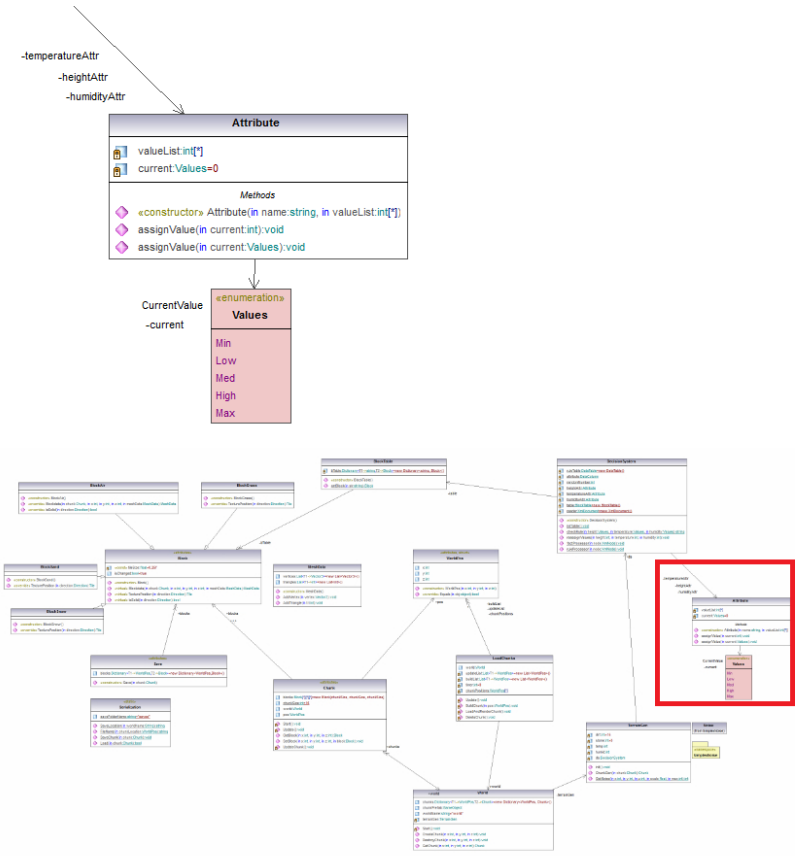

Figura E.1: Modelo completo - http://bit.ly/2cvRd8F

Apêndice F

Modelo UML da aplicação parte 6.

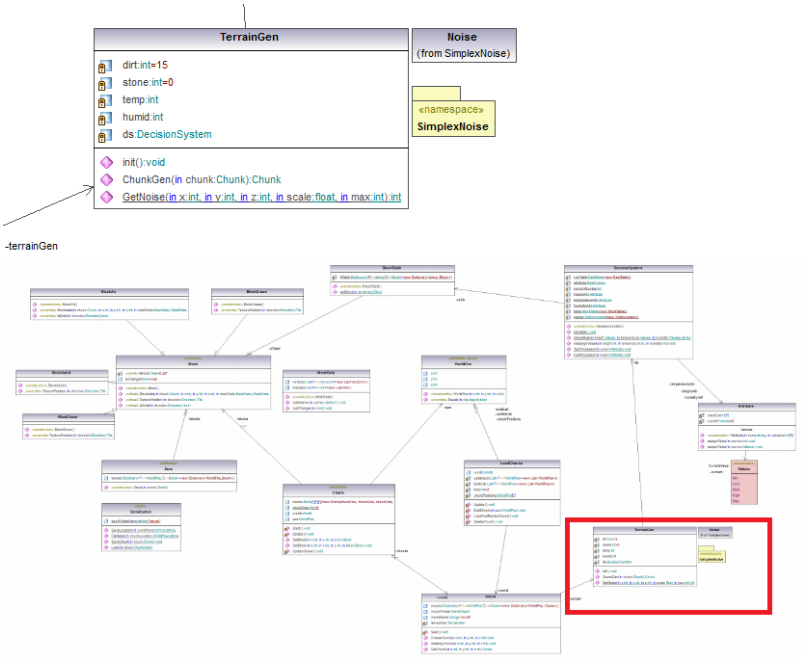

Figura F.1: Modelo completo - http://bit.ly/2cvRd8F

Apêndice G

Modelo UML da aplicação parte 7.

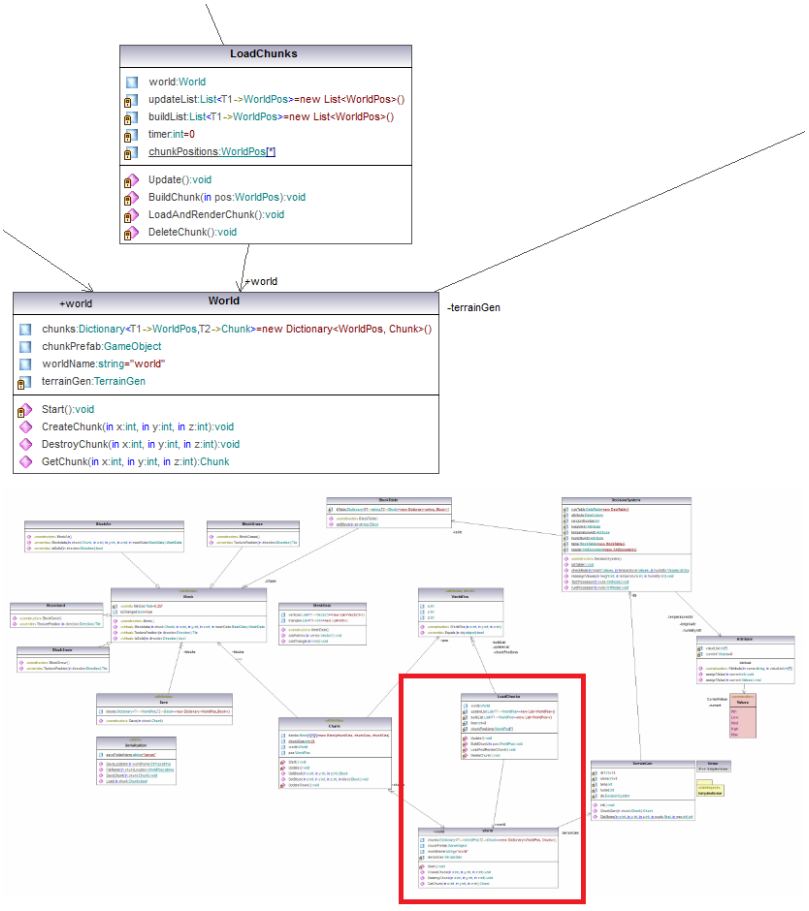

Figura G.1: Modelo completo - http://bit.ly/2cvRd8F

Apêndice H

Exemplo do processo de geração

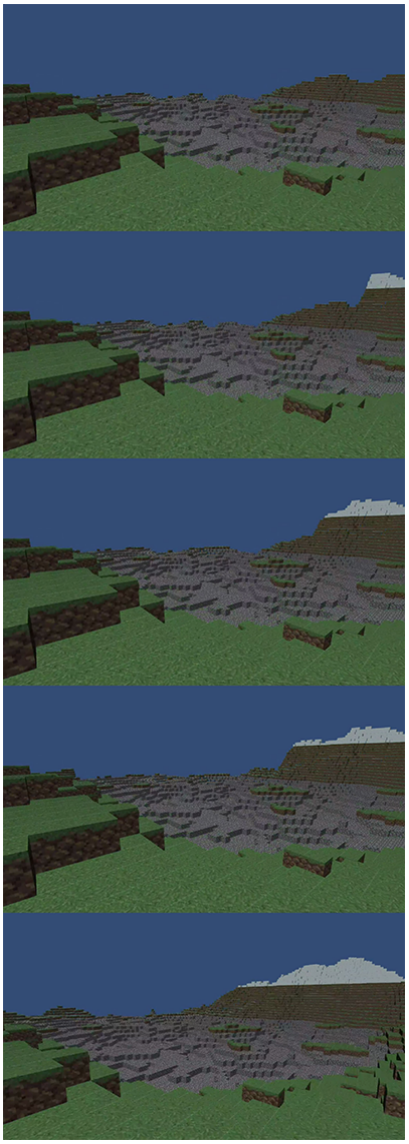

Figura H.1: Vídeo detalhado do processo - https://youtu.be/ffJKupq_TIY

Bibliografia

[1] Clips. http://clipsrules.sourceforge.net/. Accessed: 10-02-2016.

[2] How do minecraft biomes work? http://gaming.stackexchange.com/questions/26531/how-do-minecraft-biomes-work. Accessed: 13-02-2016.

[3] Id3 decision tree algorithm. http://www.codeproject.com/Articles/5276/ID-Decision-Tree-Algorithm-in-C. Accessed: 26-06-2016.

[4] Ikvm.net. http://www.ikvm.net/. Accessed: 07-05-2016.

[5] Ironpython - the python programming language for the .net framework. http://ironpython.net/. Accessed: 07-01-2016.

[6] Linq. https://msdn.microsoft.com/en-us/library/bb308959.aspx. Accessed: 26-02-2016.

[7] Minecraft. https://minecraft.net/en/. Accessed: 15-02-2016.

[8] Modelo uml da aplicaçao. http://i.imgur.com/YyyySEh.png.

[9] Pyke - python knowledge engine. http://pyke.sourceforge.net/. Accessed: 07-01-2016.

[10] Rule based systems - University of Stirling. http://www.cs.stir.ac.uk/courses/CSC9T6/lectures/3%20Decision%20Support/1%20-%20Rule%20Based%20Systems.pdf. Accessed: 09-12-2015.

[11] Simplex noise - Improved Perlin Noise Generator in C#. https://code.google.com/archive/p/simplexnoise/. Accessed: 15-03-2016.

[12] Swi-pl. http://swi-prolog.org/. Accessed: 10-05-2016.

[13] Swi-pl c#. http://www.lesta.de/prolog/swiplcs/Generated/Index.aspx. Accessed: 10-02-2016.

[14] Tennis for two - the first videogame? https://www.bnl.gov/about/history/firstvideo.php. Accessed: 06-12-2015.

[15] tuprolog. http://apice.unibo.it/xwiki/bin/view/Tuprolog/. Accessed: 26-04-2016.

[16] tuprolog Manual. https://bitbucket.org/tuprologteam/tuprolog/downloads/tuprolog-guide-3.0.pdf. Accessed: 02-05-2016.

[17] Unity 3d. https://unity3d.com/. Accessed: 15-02-2016.

[18] Unity API - Asset Workflow. https://docs.unity3d.com/Manual/AssetWorkflow.html/. Accessed: 09-02-2017.

[19] Unity API - GameObject. https://docs.unity3d.com/ScriptReference/GameObject.html. Accessed: 28-02-2016.

[20] Unity API - MonoBehaviour. https://docs.unity3d.com/ScriptReference/MonoBehaviour.html. Accessed: 28-02-2016.

[21] Unity API - Prefab. https://docs.unity3d.com/Manual/Prefabs.html. Accessed: 28-02-2016.

[22] Unity API - Start. https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html. Accessed: 28-02-2016.

[23] Unity API - Update. https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html. Accessed: 28-02-2016.

[24] Voxel tutorial. http://alexstv.com/index.php/category/voxel-tutorial. Accessed: 26-02-2016.

[25] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Artif. Intell.*, 72(1-2):81–138, January 1995.

[26] Miguel Monteiro de Sousa Frade. Genetic terrain programming. Master's thesis, Universidad de Extremadura, 2008.

[27] Azeb Bekele Eshete. Integrated case based and rule based reasoning for decision support. 2009.

[28] Petter Fogelqvist. Verification of completeness and consistency in knowledge-based systems: A design theory. 2011.

[29] Viliam Lisý, Branislav Bošanský, and Michal Pěchouček. Anytime algorithms for multi-agent visibility-based pursuit-evasion games. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 3*, AAMAS '12, pages 1301–1302, Richland, SC, 2012. International Foundation for Autonomous Agents and Multiagent Systems.

[30] Abdel-Illah Mouaddib and Shlomo Zilberstein. Optimal scheduling of dynamic progressive processing. In *ECAI*, pages 499–503, 1998.

[31] J.R. Rabunal, J. Dorado, and A.P. Sierra. *Encyclopedia of Artificial Intelligence*. Encyclopedia of Artificial Intelligence. Information Science Reference, 2009.

[32] Ruben M Smelik, Klaas Jan de Kraker, Tim Tutenel, Rafael Bidarra, and Saskia A Groenewegen. A survey of procedural methods for terrain modelling. In *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*, 2009.

[33] Gillian Smith. Procedural content generation: An overview. 2015.

[34] Julian Togelius, Noor Shaker, and Mark J. Nelson. Fractals, noise and agents with applications to landscapes. In *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, pages 57–72. Springer, 2016.

[35] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, 17(3):73, 1996.

[36] Shlomo Zilberstein and Eric Hansen. Monitoring and control of anytime algorithms: A dynamic programming approach. *Artificial Intelligence*, (126):129–157, 2001.