Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Caio Souza Fonseca

GERAÇÃO PROCEDIMENTAL DE CAVERNAS PARA JOGOS DIGITAIS 2D NA UNREAL ENGINE

RECIFE, 2016

Caio Souza Fonseca

GERAÇÃO PROCEDIMENTAL DE CAVERNAS PARA JOGOS DIGITAIS 2D NA UNREAL ENGINE

Trabalho de Graduação apresentado para o curso de Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Geber Lisboa Ramalho (glr@cin.ufpe.br)

RECIFE, 2016

Caio Souza Fonseca

GERAÇÃO PROCEDIMENTAL DE CAVERNAS PARA JOGOS DIGITAIS 2D NA UNREAL ENGINE

Trabalho de Graduação apresentado para o curso de Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco para obtenção do grau de Bacharel em Ciência da Computação.

Recife, ____ de Julho de 2016

Banca Examinadora

_____

Prof. Geber Lisboa Ramalho

(Orientador)

_____

Giordano Ribeiro Eulalio Cabral

(Avaliador)

Agradecimentos

Agradeço primeiramente à Deus, por me fornecer todas as oportunidades que eu tive e tenho, oportunidades que me permitiram trabalhar em ótimos locais, ter uma experiência de intercambio, conhecer grandes amigos e finalmente estar onde estou hoje, concluindo este curso. Pela força e coragem que me foram dadas em momentos de fraqueza, perseverança em momentos de desistência e alegria em momentos de tristeza.

À minha família em geral, por me dar condições e apoio físico e emocional por toda esta estrada, seja por incentivo, palavras, demonstrações de amor ou orações. Aos meus pais que acreditaram em minhas capacidades e sempre estiveram ao meu lado. A minhas irmãs pela companhia, amizade e momentos de alegria.

Agradeço em especial à minha amada Priscila, por me trazer alegria e força em momentos difíceis e sempre estar ao meu lado me incentivando e ajudando a ver os desafios por outros ângulos.

Aos meus amigos, alguns em especial como Luiz Fernando Sotero, Anderson Luiz Freitas Ribeiro e José de Arimatea Rocha Neto pela suas amizades e valiosos conselhos. Finalmente, à Geber Lisboa Ramalho, meu orientador que me deu ampla liberdade e apoio no desenvolvimento deste trabalho.

Resumo

A Geração Procedimental de Conteúdo, ou GPC, tem sido utilizada como uma forma de reduzir custo e tempo no desenvolvimento de jogos digitais, principalmente ao permitir a criação automática de conteúdo que outrora seria feito manualmente por artistas e *level* designers. Um dos processos mais custosos no desenvolvimento de jogos é a criação de mapas e pensando nisto, este trabalho propõe um sistema de criação de cavernas procedimentais 2D, com aspectos semelhantes a cavernas reais, para jogos. Para tanto, foram estabelecidas seis etapas para alcançar este objetivo, sendo elas: A utilização do algoritmo de Autômatos Celulares, para criar o contorno inicial da caverna; a detecção das regiões criadas, removendo as indesejadas; a conexão das regiões restantes, garantindo acesso entre todas as regiões; a utilização de *Voxels*, que representam as unidades mínimas de volume da caverna; a utilização do algoritmo de Quadrados Marchantes, com objetivo de suavizar o contorno da caverna; a detecção de arestas e criação de um muro em todo o contorno da caverna, limitando a movimentação do jogador.

Palavras-chave: Geração Procedimental de Conteúdo, Jogos, *Game Design*, Criação de mapas, Cavernas, *Voxels*, Quadrados Marchantes.

Abstract

The Procedural Content Generation, or PCG, has been used as a way of reducing costs and development time for digital games, mostly by allowing the automatic content creation that would be instead made manually by artists and level designers. One of the costliest processes in game development is the map creation and thinking of that, this work proposes a system of 2D procedurally generated caves, with similar aspects to that of real caves, for games. We established six steps to accomplish this objective, being them: The usage of the algorithm of Cellular Automata, to create the cave's initial contour; The created regions detection, removing the uninteresting ones; The connection of the remaining regions, ensuring access throughout all regions; The usage of Voxels, to represent the minimal volume unit of the cave; The usage of the algorithm Marching Squares, to make the cave's contour smoother; The edge detection and creation of a wall, limiting the player's movement.

Keywords: Procedural Content Generation, Games, Game Design, Map Creation, Caves, Voxels, Marching Squares.

Sumário

Lista de Figuras

Lista de Siglas e Abreviaturas

GPC	Geração Procedimental de Conteúdo
RPG	*Role Playing Game*
Wiki	*Wikipedia*
BSP	*Binary Space Particioning*
PbVI	Probabilidade de Vida Inicial
LimVV	Limite de Vizinho Vivos
LimVM	Limite de Vizinhos Mortos
PasR	Passage Radius
UE4	Unreal Engine 4

1 Introdução

Este capítulo está organizado em três partes, sendo elas: Um detalhamento do contexto do trabalho e sua motivação; O objetivo que buscou ser alcançado; uma breve descrição da estrutura deste documento.

1.1 Contexto e motivação

1.1.1 Geração Procedimental de Conteúdo

A indústria de jogos digitais teve um crescimento surpreendente nas últimas décadas, crescimento tal, que tornou o mercado vasto e altamente competitivo. Hoje alguns jogos chegam a ultrapassar custos que só eram imaginados para industrias como a do Cinema. O jogo Grand Theft Auto V, desenvolvido pela Rockstar em 2015, por exemplo custou cerca de 265 milhões de dólares para ser desenvolvido e alcançou 1 bilhão de dólares em cópias vendidas em três dias após o lançamento, em contrapartida o filme Avatar custou cerca de 280 milhões de dólares para ser desenvolvido e alcançou 1 bilhão de dólares no seu décimo sétimo dia após o lançamento [1]. Com valores como este, podemos ver que o mercado tem se tornado um grande risco para os desenvolvedores, já que ao mesmo tempo que um jogo pode ser um grande sucesso como Grand Theft Auto V, um jogo falho pode se tornar um prejuízo multimilionário, de forma que muitas empresas fecharam e estão fechando devido a tais prejuízos [2], como por exemplo a THQ, declarou falência em 2013.

Buscando a redução do custo de criação de conteúdo, a redução de tempo de produção, espaço em disco, rejogabilidade e possibilidades impensadas quanto a criação de conteúdo, ao mesmo tempo que fornece maneiras, outrora impossíveis, a Geração Procedimental de Conteúdo (*Procedural Content Generation*) ou GPC foi criada [3].

A GPC trata-se da criação de conteúdo para jogos de forma automática utilizando algoritmos. Estes algoritmos expressam regras claras que devem ser seguidas a partir de uma

semente (*seed*), esta irá definir o comportamento de cada procedimento do algoritmo, sendo assim, é possível alcançar sempre o mesmo resultado ao utilizar uma determinada semente [4].

O conteúdo criado por GPC é somente limitado pelas habilidades dos desenvolvedores [5], e este pode ser utilizado para a criação de qualquer conteúdo que será utilizado no jogo, este processo pode ser utilizado para criação de mapas, armas, inteligência artificial, vegetação e muitos outros tipos de recursos que serão utilizados no processo de criação do jogo [4], um ótimo exemplo de utilização ampla de GPC é o jogo No Man's Sky, desenvolvido pela Hello Games e com o lançamento previsto para 2016, jogo espacial em que o jogador se encontra em um universo fictício, em que todo um universo e o conteúdo encontrado nele é criado com a utilização de GPC, desde sistemas solares e planetas até mesmo criaturas e os sons que elas produzem [6].

Um dos primeiros jogos que apresentou uma boa utilização de GPC é o jogo *Rogue,* produzido por Michael Toy e Glenn Wichman em 1980, neste jogo o jogador assume comando de um típico aventureiro que se encontra dentro de um calabouço criado procedimentalmente [3].

1.1.2 Geração Procedimental de Conteúdo e Mapas

A construção do mapa é uma etapa crucial no desenvolvimento de jogos, este deve ser cuidadosamente criado, pois é um dos fatores que define se os jogadores irão ou não gostar do jogo, já que nele estarão presentes todos os elementos que proporcionarão diversão e desafios. Ao mesmo tempo, a criação do mapa é uma das etapas mais que mais demanda custo e trabalho, devido a toda a gama de conteúdo, arte e game design que estão inclusas em seu desenvolvimento. Por isso, é fácil ver que a GPC também pode ser utilizada para a criação de mapas [4], como por exemplo em Minecraft, desenvolvido pela Mojang em 2009, com sem mundo virtualmente infinito, No Man's Sky, com todo um universo, e The Binding of Isaac, desenvolvido por Edmund McMillen e Florian Himsl em 2011, em que todo o mapa do jogo consiste em salas procedimentais interligadas.

Em jogos com mapas de mundo aberto (onde o jogador pode escolher seu caminho, sem a necessidade de seguir o roteiro do jogo), com elementos de RPG (*Role-Playing Game*, também conhecido como jogo de interpretação de papeis) ou *Rogue-likes* (O jogo se passa geralmente em calabouços, que se tornam mais desafiadores a cada nível passado e quando o jogador morre todo o progresso é geralmente perdido), GPC pode ser e é muito utilizada para além de diminuir custos e tempo de produção [2], trazer rejogabilidade(A possibilidade de o mesmo jogador jogar várias vezes a mesma etapa do jogo sem sentir que está realmente repetindo o jogo).

1.1.3 Motivação

Embora a área de GPC já esteja em estudo a bastante tempo [3], nós vemos nas sessões anteriores que a técnica cobre uma área extensa de possibilidades de uso, mas quando falamos de geração de cavernas 2D com aspectos semelhantes ao de cavernas reais, que possam ser utilizadas em jogos, as soluções apresentadas deixam a desejar. Como explicaremos posteriormente nos capítulos 2 e 3, as técnicas que são comumente utilizadas e as soluções propostas não são satisfatórias, pois trazem resultados que lembram obras humanas e não da natureza.

1.2 Objetivo

O objetivo geral deste trabalho é construir um sistema de criação de cavernas 2D, com aspectos semelhantes a cavernas reais, para jogos, e que possam ser utilizadas em projetos futuros. Como objetivos específicos, nós temos:

- Acelerar o processo de desenvolvimento de jogos, reduzir custos, tempo, quantidade de trabalho e aumentar a rejogabilidade ao possibilitar a criação de parte do mapa ou ele todo de forma automática e com as infinitas possibilidades que GPC proporciona.
- Explorar técnicas comuns de criação de cavernas.

Para alcançá-los, algoritmos como o Particionamento Binário de Espaço (BSP), Autômatos Celulares e outros serão pesquisados e estudados, com a finalidade de encontrar a melhor solução para a criação da caverna 2D, com múltiplas salas e um contorno semelhante ao encontrado na natureza, além da utilização do algoritmo de Quadrados Marchantes (*Marching Squares*), para a suavização do contorno.

Propomos então a utilização de Autômatos Celulares como etapa inicial, seguida de cinco etapas de processamento do mapa gerado, para nos aproximarmos mais de um resultado semelhante ao encontrado no mundo real e que possa ser utilizado em jogos.

Como parte da solução, nós iremos aplicar a solução criada na Unreal Engine 4, UE4, demonstrando não somente a possibilidade real da aplicação, como também testando e avaliando o sistema de Visual Scripting da UE4 chamado Blueprints.

1.3 Estrutura do Documento

Os próximos capítulos deste trabalho serão organizados da seguinte maneira:

- Capítulo 2: Dá um aprofundamento em geração de conteúdo procedimental, explicando e mostrando exemplos sobre os tipos de conteúdo que podem ser criados, as vantagens e desvantagens em relação à criação de conteúdo de forma manual e as dificuldades presentes.
- Capítulo 3: Fala um pouco mais sobre a motivação da utilização de GPC para criação de cavernas e apresentamos algumas técnicas que são comumente utilizadas para criá-las.
- Capítulo 4: Descreve a solução proposta, dando ênfase à técnica e às etapas que foram tomadas para a criação da caverna.
- Capítulo 5: Apresenta os resultados alcançados e falamos sobre possíveis otimizações.
- Capítulo 6: Finaliza o trabalho com as nossas considerações finais e propostas de trabalhos futuros.

2 Geração Procedimental de Conteúdo

Será apresentado neste capitulo o conceito da Geração Procedimental de Conteúdo de forma mais detalhada. Posteriormente, iremos mostrar a diversidade de conteúdo que pode ser criado utilizando GPC, deixando claro a versatilidade presente em sua utilização. Seguindo adiante, falaremos mais especificamente das diferenças, vantagens e desvantagens de usar GPC e dos métodos manuais. Finalmente veremos as principais dificuldades, bem como alguns exemplos de boa e má utilização GPC.

2.1 Conceito

De acordo com Togelius et al (2011), GPC em jogos se refere a criação de conteúdo de forma automática utilizando algoritmos. Já Hendrikx et al (2011), afirma que GPC é a aplicação de computadores para gerar conteúdo de jogos, distinguir instâncias interessantes entre os resultados gerados e selecionar instâncias interessantes em prol dos jogadores.

Estes algoritmos utilizam regras bem definidas que seguem uma semente (variável do tipo inteiro), esta pode ser reutilizada posteriormente e gerar o mesmo resultado, além de um certo grau de aleatoriedade e pseudo-aleatoriedade para gerar os mais diversos resultados. Sendo assim, estão somente restritos à habilidade do programador e às próprias restrições do jogo em questão, é então fácil compreender que GPC pode ser utilizado para as mais diversas áreas, inclusive sendo utilizado para não somente gerar um conteúdo por completo, mas também para servir de passo inicial no desenvolvimento [9]. É também importante dizer que estes algoritmos quando aplicados diretamente durante o jogo, e não somente para a criação de um conteúdo que pode ser utilizado posteriormente, podem ser aplicados tanto tem tempo real, como em Minecraft, No Man’s Sky e Terraria, desenvolvido pela Re-Logic em 2011, em que o mapa é criado em tempo de execução, enquanto o jogador anda pelo cenário, como também pode ser aplicado de forma pré-carregada como em The Binding of Isaac,

Rogue Legacy, desenvolvido pela Cellar Door Games em 2013, em que o mapa é criado no tempo de carregamento.

Entretanto, esta técnica não é de fácil utilização [5], além do fato de o computador necessitar de certa capacidade computacional para executar os algoritmos [8], que podem ser bastante complexos, existe também o fato de que por melhores que sejam os resultados em geral do algoritmo, sempre existe a possibilidade de alcançar um resultado válido, mas que não seja interessante para o jogador.

Podemos então verificar o que os desenvolvedores buscam GPC com alguns objetivos em mente, geralmente eles são a redução do tempo de produção de conteúdo, acarretando na redução dos custos, redução do espaço em disco do jogo, já que o conteúdo não está propriamente criado e guardado, mas é na verdade um resultado dos resultados do algoritmo e a possibilidade de alcançar resultados mais criativos através da experimentação que GPC proporciona[3].

2.2 GPC vs Método Manual

O método mais comum de produção de conteúdo na indústria de desenvolvimento de jogos é o Hand Crafted, ou feito à mão, ou seja, cada recurso que será utilizado no jogo é desenvolvido etapa a etapa pela interação humana. Se formos olhar a criação de uma espada para um personagem, teremos algumas etapas que costumam estar presentes no ciclo de criação. Estas etapas são: Criação do conceito (desenhos rápidos feitos por artistas digitais); modelagem High Poly (criação do modelo 3D com todos os detalhes); modelagem Low Poly (simplificação do modelo 3D); criação dos mapas de UV (Normais, Ambiente Oclusion, etc); texturização; rigging e animação [10]. Como podemos ver, para a criação de um único recurso que será utilizado no jogo, nós temos múltiplas etapas que passarão por vários especialistas, por isso o preço elevado. Levando em consideração este exemplo da espada, utilizando GPC é possível a criação, tanto do zero, como ao mesclar vários recursos previamente feitos a

mão, como por exemplo dividir a espada em três partes, lamina, guarda e punhal, criar cinco variações de cada parte e utilizar GPC para criar todas as possíveis combinações.

Em jogos como World of Warcraft, desenvolvido pela Blizzard no ano de 2004, em que a estimativa de custo para produção do jogo original gira em torno de vinte milhões de dólares a cento e cinquenta milhões de dólares, cerca de 30 a 40% é dedicado a produção de conteúdo [8]. Sendo assim, vemos como métodos automáticos são interessantes, principalmente para pequenas empresas, como estúdios *Indie*, em que GPC permite a possibilidade de competição com empresas maiores [11].

Ao compararmos os dois métodos, fica claro perceber que ambos possuem vantagens e desvantagens, bem como quando é melhor usar uma ou a outra. Podemos então detalhar algumas delas na tabela a seguir.

Quadro 1 – GPC vs Hand Crafted

GPC		**Hand Crafted**	
Vantagens	**Desvantagens**	**Vantagens**	**Desvantagens**
Possibilidade de ser implementado com poucos desenvolvedores e artistas	Mudanças, mesmo pequenas, podem ser trabalhosas	Mudanças pequenas como adicionar árvores em frente a uma caverna requer menos trabalho	Alto custo
Acesso a infinitos resultados sem a necessidade de mais trabalho, uma vez que o algoritmo já está feito	Precisa de certo poder computacional, principalmente quando utilizado em tempo real,	Alta liberdade para criação de detalhes	Longo tempo de produção
Reduz a quantidade de trabalho	Pode ter resultados desinteressantes	Produto final fiel ao conceito	Requer vários desenvolvedores e artistas
Acelera o processo de desenvolvimento	A complexidade de desenvolvimento aumenta rapidamente		
Possibilidades virtualmente infinitas	Produto final pode ser bastante diferente do desejado		
Aumenta a rejogabilidade	É fácil criar conteúdo repetitivo		
Altamente versátil			
Reduz custos			

Fonte: Elaborado pelo autor

2.3 Tipos de Conteúdo

Hendrikx et al (2011) explica em seu trabalho alguns tipos conteúdo que podem ser criados utilizando GPC e os apresenta de forma detalhada com a seguinte imagem.

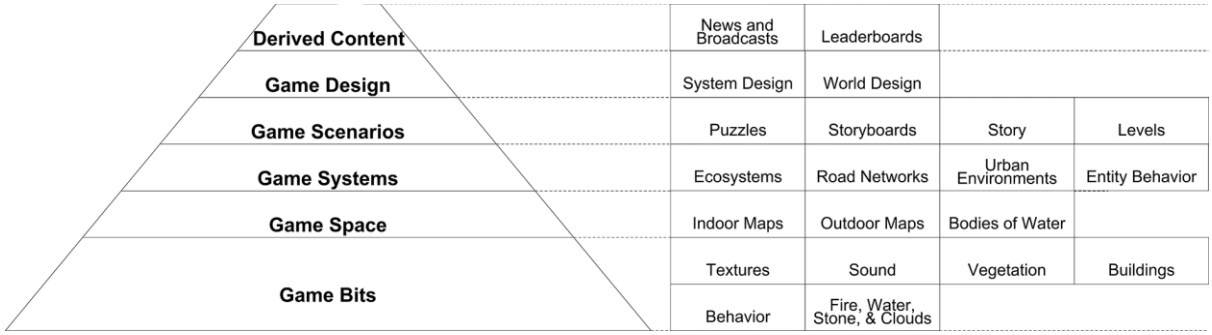

Figura 1. Alguns tipos de conteúdo gerados por GPC (HENDRIKX et al, 2011, adaptado pelo autor)

Nesta imagem vemos que ele divide os conteúdos em seis categorias:

- Conteúdo Derivado: São itens criados para aumentar o sentimento de imersão do jogador no mundo do jogo, como quadros de pontuações, avisos do jogo, etc.
- Game Design: São partes dos jogos que contém as regras e objetivos do jogo, bem como o tema, estória, aspectos gráficos, etc.
- Cenários: Definem a ordem de como os eventos do jogo devem ser dadas, podendo ser por quebra-cabeças, estórias, níveis, etc.
- Sistemas: Consistem dos sistemas do jogo que definem regras do ambiente em que o jogador está, como simulação de cidades, movimentos em ruas, etc.
- Espaço: É exatamente o espaço do ambiente em que o jogador está presente, podendo ser florestas, labirintos, planícies, cidades, etc.
- Bits: São unidades elementais de conteúdo do jogo, sejam elas interativas ou não. Aqui estão texturas, sons, vegetação, construções, personagens, armas, etc.

Neste trabalho, o resultado final pode ser caracterizado como o Espaço do jogo, pois a caverna gerada pode ser tanto o mapa total do nível, como também pode ser adaptada para ser apenas uma sala ou uma caverna englobada em um mapa maior.

2.4 Exemplos de GPC em jogos

Com o conhecimento da viabilidade da utilização de GPC e de sua versatilidade como visto acima, separamos alguns exemplos de como GPC já foi utilizada na indústria de jogos, destacando boas e más utilizações da técnica.

Embora o jogo No Man’s Sky seja atualmente uma das melhores referências quando falado de GPC em jogos, este não será citado como uma boa utilização de GPC, pois o jogo ainda não teve seu lançamento oficial até a data deste trabalho. Então não é possível afirmar que o prometido e visto em vídeos e entrevistas foi realmente cumprido.

2.4.1 Minecraft

Minecraft utiliza *Voxels* e GPC para criar um mundo virtualmente infinito e único para cada jogo novo. O mapa é construído por blocos de *Voxel* de um metro cúbico, ligados uns nos outros, seguindo uma série de regras [6]. Mais especificamente ruídos *Perlin* para a criação da topografia, *Perlin Worms* para a criação de cavernas e mais [12]. Este mundo só é realmente gerado no jogo quando há demanda, de forma que um *Voxel* que não está visível ao jogador não é renderizado. É interessante apontar que em Minecraft as regras que ditam a criação do mapa, também definem biomas juntamente com suas respectivas criaturas, vegetação, etc.

A figura a seguir mostra parte de um mundo de Minecraft com seus biomas representados com cores.

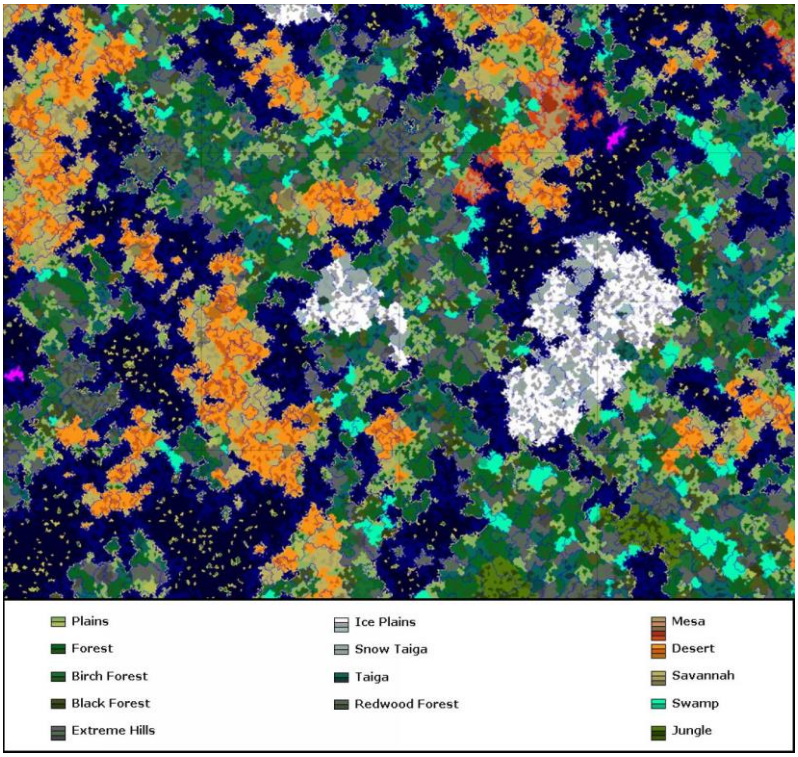

Figura 2. Mapa com biomas de Minecraft [25].

2.4.2 Borderlands 1 e 2

Borderlands 1 e 2, desenvolvidos pela Gearbox Software nos anos de 2009 e 2012, respectivamente, jogos de tiro em primeira pessoa com elementos de RPG e utilizam GPC para a criação de armas, a técnica utilizada é a de combinar partes predefinidas para gerar as armas. Ambos os jogos se tornaram icônicos por sua gigante gama de armas, de acordo com a própria *Wiki* de Borderlands [13], o jogo possui cerca de 17.750.000 variações diferentes de armas, no caso de Borderlands 2 esse número não é público. Estas armas seguem oito categorias, sendo elas pistolas repetidoras, revolveres, submetralhadoras, rifles de combate, escopetas, rifles de longo alcance, lança misseis e armas Eridianas. Além disso, cada arma possui sua raridade, efeito elemental, produtora e os componentes, ou partes, que as formam.

Abaixo na figura é possível ver alguns exemplos de armas com suas respectivas características.

Figura 3. Armas em Borderlands [26].

2.4.3 The Binding of Isaac

The Binding of Isaac é um jogo em que o jogador controla Isaac por uma série de calabouços gerados com GPC, de forma semelhante ao icônico Rogue. No jogo, o jogador ganha recompensas por explorar as salas do calabouço, luta com inimigos e eventualmente luta contra o chefe do nível do calabouço, após vencer o chefe o jogador é lançado no próximo nível que será mais difícil que o anterior. Uma vez que o jogador é derrotado, todo o progresso do jogo é perdido.

Estes níveis são criados seguindo algumas regras, no caso cada mapa consiste de quatro salas primárias e de cinco a vinte salas secundárias. As quatro salas principais são: A sala do tesouro; A sala do chefe; A sala da loja; A sala secreta. Estas salas sempre serão geradas em cada nível do calabouço. As salas secundárias são arenas de combate e além delas também existem salas de contexto que podem ser geradas de acordo com o nível do calabouço, estas podem ser de combate, desafios, quebra-cabeças, armadilhas, *arcades* e mini chefes [14].

Nas figuras a seguir podemos ver um exemplo do mapa de um nível do calabouço de The Binding of Isaac e em seguida o significado dos ícones das salas.

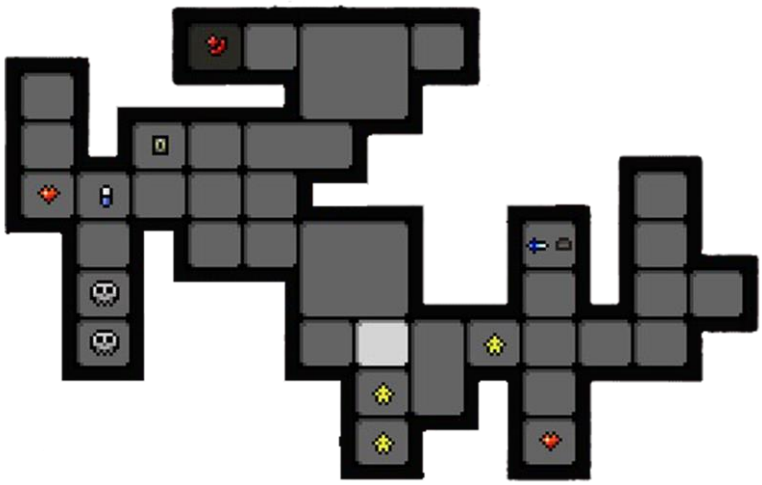

Figura 4. Mapa de The Binding of Isaac [27].

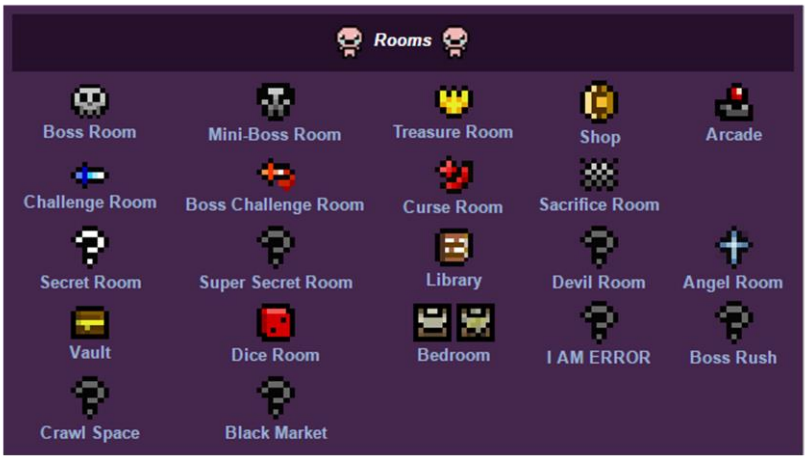

Figura 5. Legenda dos icones das salas de The Binding of Isaac [28].

2.4.4 Spore

Spore, desenvolvido pela Maxis no ano de 2008, utiliza GPC para criar todas as criaturas, suas animações e habilidades em tempo real, estas criaturas estariam armazenadas em arquivos na escala dos *Megabytes* ou *Gigabytes* e graças a GPC eram utilizados *Kilobytes* de espaço em disco [30]. A ideia de Spore é que dado alguns elementos finais das criaturas, como bocas, patas, pernas, braços, etc. O jogador pode escolher as partes desejadas e uma criatura será criada. Esta criatura terá suas animações definidas pelas partes escolhidas, assim é possível os mais variados resultados [3].

Na figura a seguir é possível ver o editor de criaturas de Spore, neste caso o jogador está ajustando as patas de sua criatura. É possível ver na esquerda as patas predefinidas, organizadas em linhas, que podem ser escolhidas e três interrogações em cada linha, que indicam que o jogador pode desbloquear mais patas naquela linha que são relacionadas com a primeira pata da linha.

Figura 6. Editor de criaturas de Spore [29].

2.5 GPC e sua avaliação

Na sessão anterior pudemos observar o sucesso de GPC em diversos exemplos de jogos e a versatilidade proporcionada pela técnica. Esta versatilidade torna, de certa forma, difícil a sua avaliação quando comparado de projeto a projeto, já que por exemplo na criação de mapas de Minecraft é interessante o alto desempenho, pelo fato de ser necessário a geração do mapa em tempo real, e em The Binding of Isaac o mapa é previamente carregado, de forma que o desempenho não é mais o foco.

Em geral uma boa solução com GPC traz a redução do tempo de produção de conteúdo, custos, espaço em disco do jogo, e a possibilidade de alcançar resultados mais criativos através da experimentação, como explicado na sessão 2.1.

No próximo capitulo iremos detalhar algumas características de um bom sistema de GPC para criação de cavernas 2D com aspectos semelhantes ao de cavernas reais.

3 GPC e Geração de Níveis

Neste capítulo, iremos abordar a utilização de GPC mais especificamente para geração de níveis. Falaremos sobre os tipos de níveis que são comumente gerados com GPC, depois iremos focar em estruturas de cavernas, descrevendo as técnicas que são mais utilizadas para gerá-las.

3.1 Tipos de Níveis

Podemos observar em alguns jogos as diversas possibilidades de GPC para criação de níveis, alguns exemplos são: Minecraft gera um mundo 3D virtualmente infinito composto por múltiplos *Voxels*; Terraria cria um mundo 2D de tamanhos fixos, também com *Voxels*; The Binding of Isaac, com seus calabouços compostos por salas primárias e secundárias; No Man's Sky com seu universo infinito, composto por galáxias, sistemas solares e planetas gerados com GPC.

Ao fecharmos o espaço de observação para níveis de tamanho fixo, começamos a ver que algumas estruturas se tornam mais comuns, elas sendo[15]:

- Calabouços
- Cavernas
- Ilhas
- Labirintos

Devido ao foco deste trabalho, nós iremos analisar somente estruturas de cavernas.

3.2 Estruturas de Cavernas

O principal motivo para a escolha de estruturas de cavernas é a versatilidade, as cavernas podem ser utilizadas como um mapa todo ou parte do mapa, podem ter múltiplas salas ou apenas uma, ser semelhante a um calabouço, labirinto ou uma caverna real e muitas

outras características que estão sujeitas ao game design e que podem ser facilmente adaptadas.

Com o objetivo de criar cavernas, nós reunimos algumas técnicas que são comumente utilizadas com GPC para criação de cavernas 2D, é importante o foco em 2D, pois cavernas 3D trazem uma complexidade maior e técnicas diferentes.

3.2.1 Propriedades de Cavernas

Primeiramente, precisamos entender qual a topologia de uma caverna real e quais propriedades são atraentes para uma solução que deve ser utilizada em jogos. Na figura abaixo, temos um mapa topográfico da caverna de Gyspum em Las Vegas.

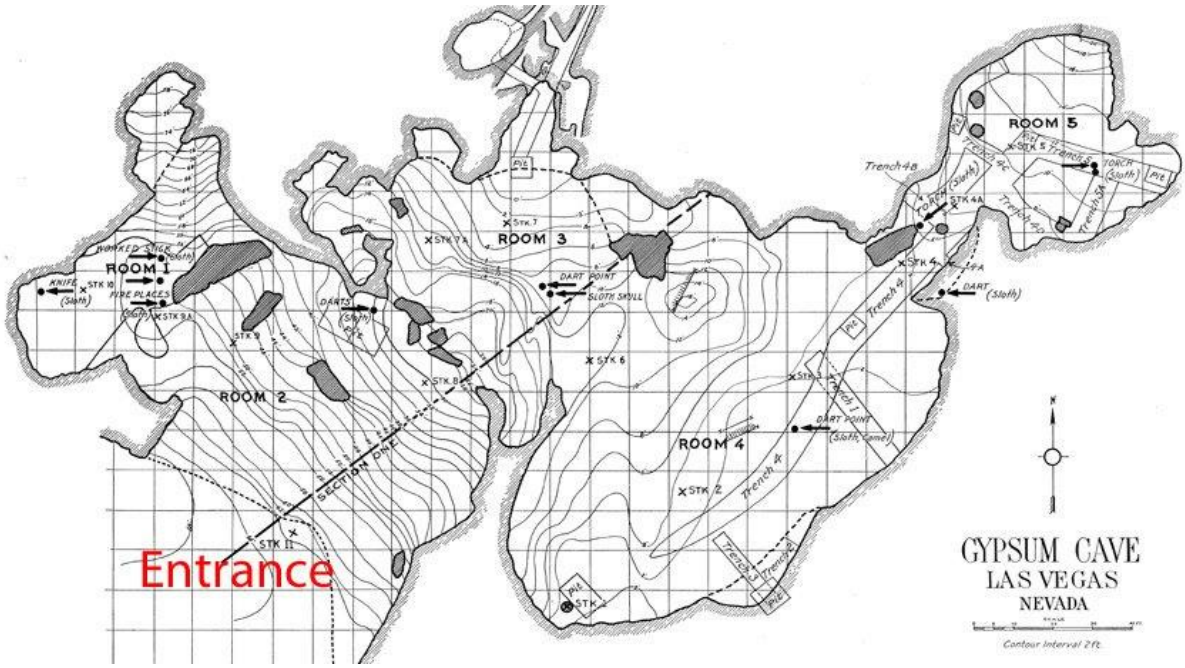

Figura 7. Contorno da caverna de Gypsum, Las Vegas. [31]

Podemos observar a existência de múltiplas salas de tamanhos diferentes, com acesso umas às outras, além disso vemos também regiões de paredes dentro das salas, como colunas, e principalmente a forma do contorno, que segue um formato natural, que lembra mais algo aleatório do que feito pelo homem. Essas são as características de uma caverna natural que queremos presentes em nossa solução.

Por se tratarem de cavernas para jogos nós precisamos ser restritos quanto aos resultados gerados, sendo assim, podemos estabelecer algumas características ideais para nossa solução:

- A caverna gerada precisa ter um contorno que se assemelhe ao encontrado na natureza, algo como a figura 7.
- A caverna será formada por salas, ou regiões, com tamanhos diferentes.
- Todas as salas precisam ser acessíveis.
- As salas precisam ter um tamanho mínimo e máximo, para não termos salas inutilizáveis ou espaços muito abertos.
- A caverna precisa possuir regiões de paredes dentro das salas, para dar a impressão de serem colunas naturais.

Com essas características definidas, nós podemos então explorar as técnicas que são comumente utilizadas para decidirmos qual delas melhor satisfaz estas características.

3.2.2 Técnicas Comuns

Ao falarmos de GPC para criação de cavernas 2D, algumas técnicas se destacam, seja por sua praticidade, versatilidade ou semelhança ao natural. Selecionamos quatro técnicas para serem estudadas com maior detalhe, estas são: Particionamento Binário de Espaço (BSP); Triangulação de Deulanay; Autômatos Celulares.

3.2.2.1 Particionamento Binário de Espaço

O Particionamento Binário de Espaço (BSP), primeiramente proposto por Fuchs et al (1980), é um algoritmo que propõe uma solução simples e elegante. A ideia é que dado uma área do mapa, o algoritmo escolhe uma direção aleatória, para fazer a divisão do mapa ser vertical ou horizontal, uma posição aleatória dentro dos limites da área, que será onde a partição ocorrerá e então a área é dividida em duas subáreas. Criando assim uma estrutura como uma árvore binária em que a cada nível da árvore são criadas duas subdivisões, é importante notar que a área inicial e a quantidade de níveis de subdivisões são definidas antes

do algoritmo ser rodado. Uma vez que o limite de níveis foi alcançado, serão criadas salas com tamanhos aleatórios dentro das subdivisões. Após todas as salas serem criadas, iremos varrer cada nível de subdivisão, começando do nível das folhas, e criaremos conexões de uma sala para a outra. Ao fim do segundo nível, nós teremos múltiplas salas de tamanhos diferentes conectadas umas às outras [15][16][17].

Nas figuras abaixo podemos ver um exemplo de como o BSP funciona.

Dada a área inicial, definimos aleatoriamente um corte vertical e também aleatoriamente a posição do corte e então dividimos a área em A e B.

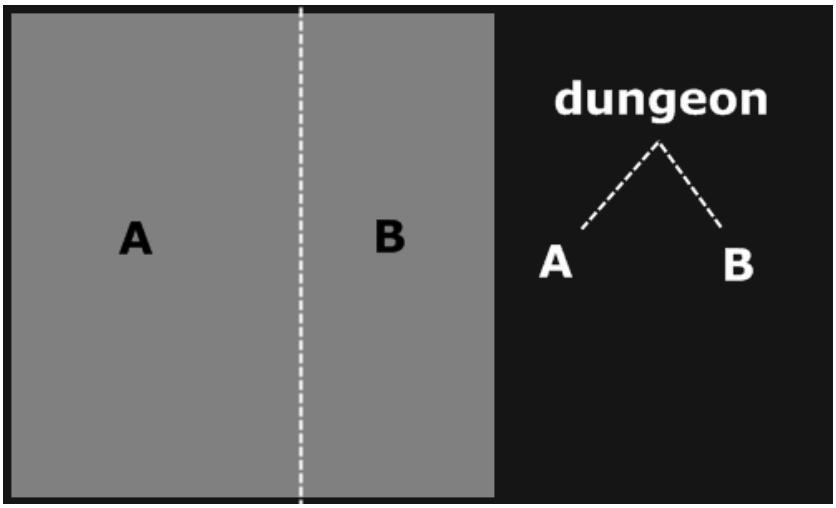

Figura 8. Primeiro nível do BSP [17].

Prosseguindo com o algoritmo, iremos agora criar da mesma forma as subdivisões das áreas A e B, criando então A1, A2 e B1, B2.

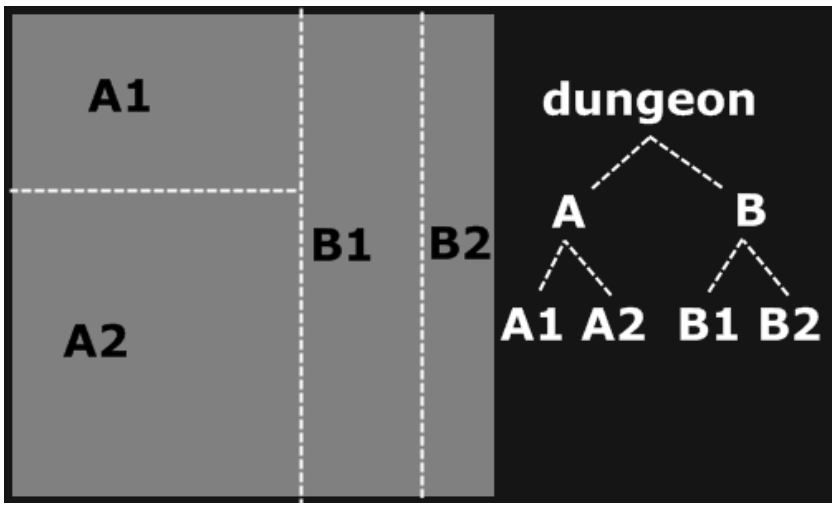

Figura 9. Segundo nível do BSP [17].

Dando continuação, ao término da subdivisão do quarto nível nós teremos algo como a figura a baixo.

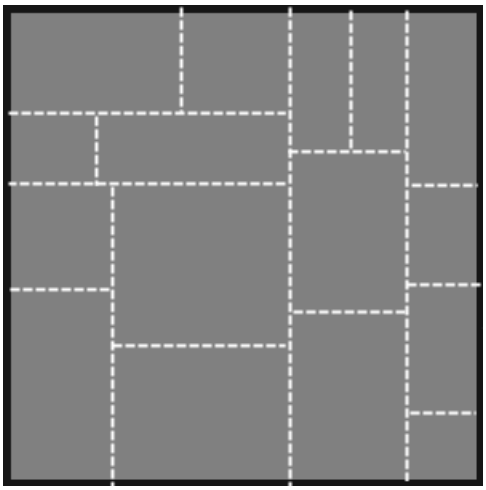

Figura 10. Após 4 níveis de subdivisões do BSP [17].

Com todas as subdivisões criadas, podemos criar então as salas em cada subdivisão.

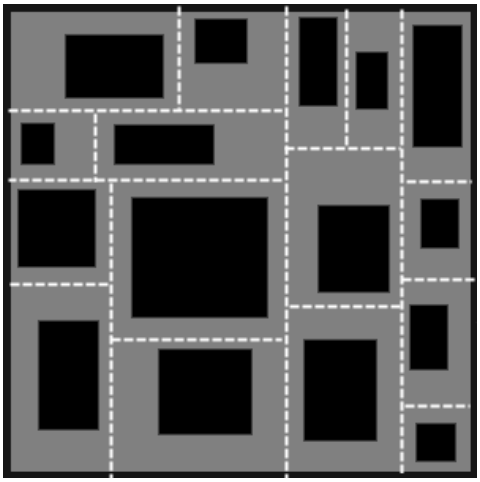

Figura 11. Criação das salas nas subdivisões do BSP [17].

Uma vez que todas as salas foram criadas, começamos a criar as conexões, primeiramente nas salas encontradas em subdivisões de nível quatro.

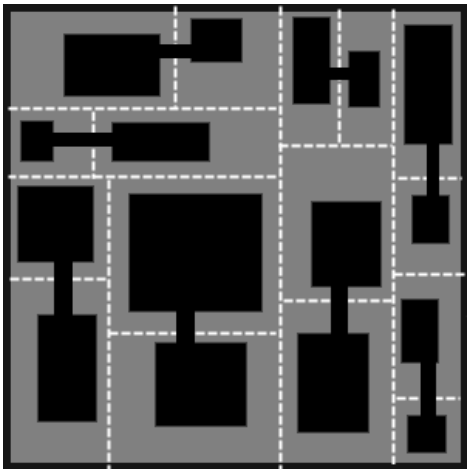

Figura 12. Conexões criadas entre as salas do quarto nível de subdivisões do BSP [17].

Dando continuação, nós criamos as conexões para as subdivisões de nível três. É importante notar que devido às conexões previamente construídas, nós garantimos que todas as salas de nível três podem ser alcançadas pelas salas de nível quatro e vice-versa.

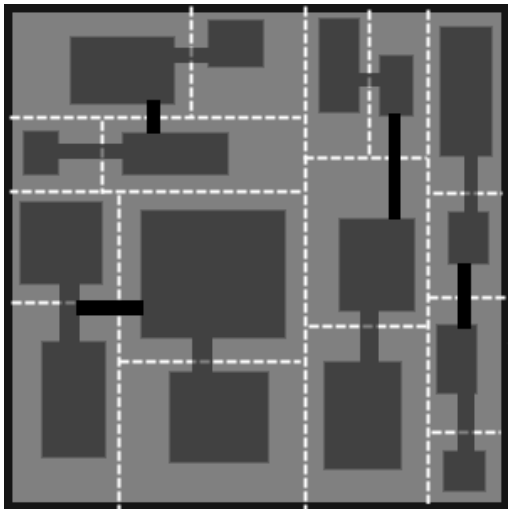

Figura 13. Conexões criadas entre as salas do terceiro nível de subdivisões do BSP [17].

Já que neste exemplo nós estamos trabalhando com quatro níveis, quando chegamos no segundo nível e criamos as conexões, nós já garantimos que todas as salas são acessíveis e conectadas entre si.

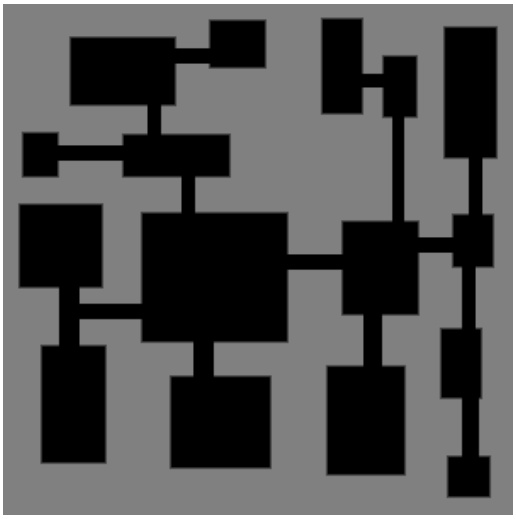

Figura 14. Conexões criadas entre as salas do segundo nível de subdivisões do BSP [17].

Com essa breve demonstração, nós podemos ver que o algoritmo é simples e traz resultados interessantes, também é fácil perceber que ele pode ser aliado com outros algoritmos para alcançar resultados mais complexos, a criação das salas poderia ser feita com autômatos celulares, por exemplo. Embora BSP seja ótimo para criação de calabouços e estruturas com salas conectadas, podemos ver que este método não satisfaz algumas das características desejadas, sendo elas o contorno natural, já que as salas e passagens são retangulares e a falta de colunas nas salas.

3.2.2.2 Triangulação de Delaunay

O método de triangulação de Delaunay foi proposto em 1934 por Boris Delaunay, é um método de triangulação de pontos muitas vezes utilizado para conectar grafos[18]. Esta técnica foi primeiramente proposta para a criação de mapas pelos desenvolvedores de TinyKeep, desenvolvido pela PhiGames em 2013. O algoritmo consiste em: A criação de salas retangulares de tamanho aleatório; Separação das salas; Seleção das salas principais, de acordo com um tamanho minimo; utiliza a triangulação de Delaunay para conectar as salas principais e então cria um grafo para representá-las; transforma-se o grafo numa árvore de extensão mínima, que garante que todas as salas são conectadas entre si e que não existe um excesso de conexões; criam-se corredores e salas secundárias, finalizando o algoritmo [19].

Na figura abaixo podemos ver o resultado final do algoritmo proposto, as salas vermelhas são as principais, as azuis são as secundárias e os corredores são os trechos branco e azul.

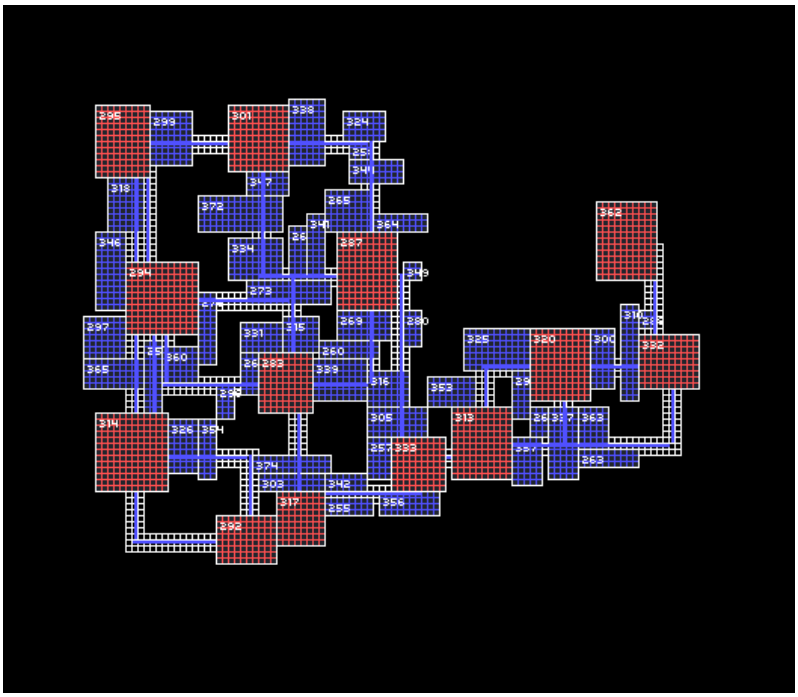

Figura 15. Resultado do algoritmo proposto pela PhiGames [19].

Embora os resultados sejam bastante interessantes, de certa forma mais interessantes do que os apresentados pela utilização de BSP, nós ainda não conseguimos satisfazer as características desejadas, a estrutura ainda não se assemelha ao natural, isto é, ainda é composta por retângulos e também não possui colunas de paredes nas salas.

3.2.2.3 Autômatos Celulares

Inicialmente proposto por Stanislaw Ulam, como uma forma de simular fluidos em 1950[15], Autômatos Celulares são hoje utilizados em diversas áreas, uma delas sendo a criação de mapas. Johnson et al (2010) anunciou a primeira proposta de utilização de Autômatos Celulares em jogos, no caso, a técnica foi utilizada para criar estruturas de mapas semelhantes a cavernas.

O Algoritmo em si é bem simples, nós temos uma probabilidade de vida inicial, PbVI, um limite de vizinhos vivos, LimVV, e um limite de vizinhos mortos, LimVM, onde vivo

corresponde a uma célula com valor 1 e morto a uma célula com valor 0 e um valor de passos de suavização, com esses parâmetros nós seguimos as seguintes etapas.

Dado o tamanho do mapa, cada posição é tratada como uma célula e ela é populada com 0 ou 1 de acordo com a PbVI, abaixo vemos um exemplo de um estado inicial do algoritmo.

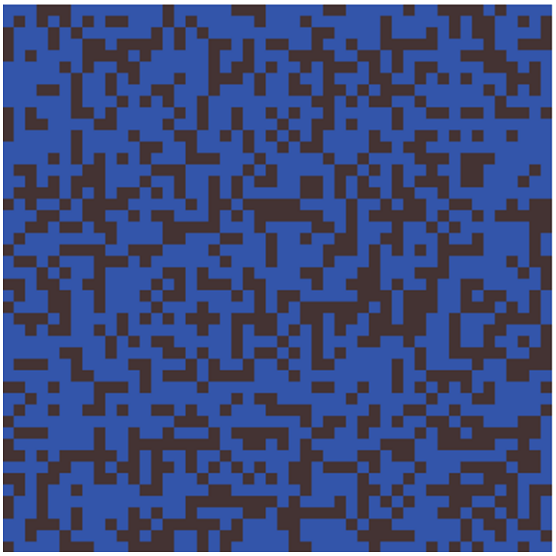

Figura 16. Primeira etapa do algoritmo de Autômato Celular [21].

A seguir começa o processo de suavização do ruído criado, que será rodado pela quantidade de passos de suavização. O algoritmo irá olhar cada célula previamente populada e analisar os seus vizinhos, contando as ocorrências de células vivas e mortas. Com o resultado desta contagem e também com os valores dos parâmetros de LimVV e LimVM a célula em questão irá mudar o seu estado. Nas próximas figuras vemos as etapas de suavização a partir do estado inicial mostrado acima.

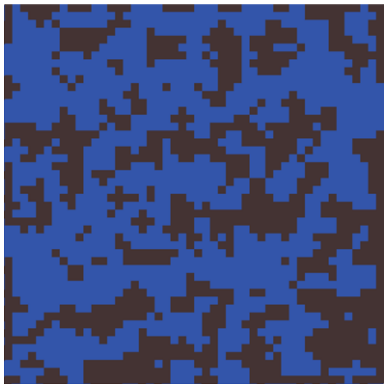

Figura 17. Primeiro passo de suavização do Autômato Celular[21].

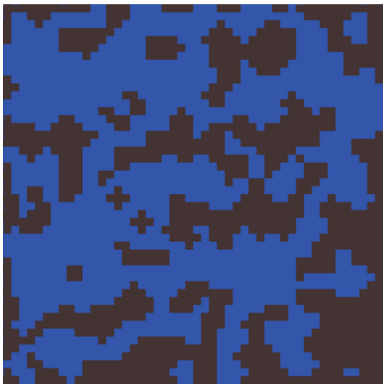

Figura 18. Segundo passo de suavização do Autômato Celular[21].

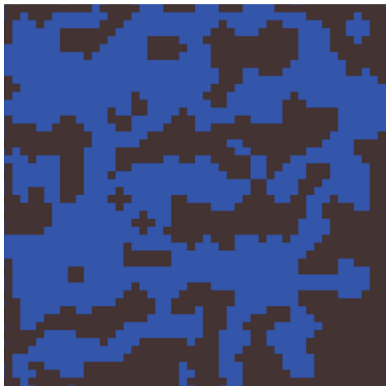

Figura 19. Terceiro passo de suavização do Autômato Celular[21].

Ao fim do algoritmo, nós temos uma estrutura aleatória com um aspecto bastante interessante e semelhante ao de uma caverna natural, como podemos ver na figura 19. É importante notar que apenas a característica de garantir o acesso a todas as regiões não foi cumprindo, como podemos ver na figura a baixo estas regiões circuladas.

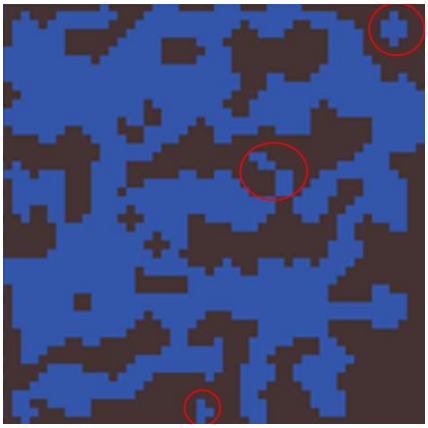

Figura 20. Regiões não conectadas do algoritmo de Autômato Celular.[21](Adaptado pelo autor)

Devido a esse bom resultado, decidimos escolher este método para nossa solução e iremos atacar especificamente essa característica não atendida de não garantir a conexão de todas as regiões.

4 Solução proposta

No decorrer deste capítulo, nós iremos descrever a solução proposta. Passaremos por cada etapa tomada na criação do mapa, mostrando as técnicas utilizadas, as adaptações que precisaram ser feitas e as dificuldade encontradas. O capítulo está dividido em cinco sessões apresentadas da seguinte forma: Primeiramente iremos abordar a implementação do Autômato Celular com suas adaptações, desde o momento inicial até o fim de seus passos de suavização, este foi o algoritmo escolhido, pois atende a maioria das características definidas na sessão 3.2.1, além de gerar uma topografia plausível para uma caverna natural, como podemos ver ao compararmos as figuras 19 e 7; Em seguida detectamos as regiões criadas e separamos em regiões de salas e muros, para as salas serem conectadas e os muros criados; Falamos então sobre o algoritmo utilizado para conectar as regiões previamente encontradas e a necessidade desta etapa; Passamos então para a criação do mapa em si, como objetos 3D, utilizando *Voxels* 2D; Iremos então, com a utilização do algoritmo de Quadrados Marchantes suavizar os *Voxels*; Finalmente nós iremos detectar as arestas da sala final e das regiões de paredes para construir um muro por todo contorno, impedindo a passagem do jogador.

4.1 Autômato Celular

Como demonstrado no capítulo anterior, o algoritmo de Autômatos Celulares consiste basicamente da etapa inicial da geração de nossa caverna, com sua utilização, nós teremos uma base para ser processada nas etapas seguintes.

Nesta etapa inicial, nós iremos primeiramente verificar se uma variável booleana “*Use Random Seed*” está verdadeira ou falsa, esta variável irá definir, como o nome indica, se utilizaremos uma semente aleatória ou uma predefinida, para mantermos os resultados consistentes, nós iremos manter esta variável desativada e utilizaremos a semente 842498. Em seguida nós inicializamos uma matriz 2D com quantidade de linhas e colunas de acordo

com o comprimento e largura do mapa desejado, cada posição da matriz será uma célula. Para cada posição da matriz, nós iremos verificar se esta posição está na borda do mapa, se estiver então nós colocamos o valor 1 na célula, caso não seja uma posição da borda, nós iremos criar um valor aleatório que será comparado com a PbVI definida antes do início do algoritmo, com tem um padrão de 50%, caso seja menor que o valor da PbVI então colocamos o valor da célula como 0, se não 1. Ao fim nós temos algo como a figura a baixo, onde os quadrados pretos são células vivas, ou seja, valor 1, e os quadrados brancos são células mortas, valor 0.

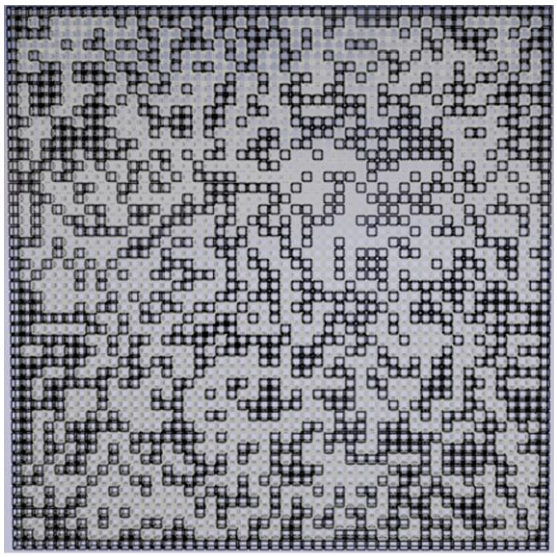

Figura 21. Etapa inicial do algoritmo de Autômato Celular na solução proposta. (Elaborado pelo autor)

Em seguida nós damos início à etapa de suavização, esta etapa irá rodar o algoritmo pela quantidade de vezes definida em uma variável que chamamos de "*Smoothing Steps*", com o valor padrão de 5. Para cada passo de suavização, nós iremos observar cada célula do mapa, contamos a quantidade de vizinhos ativos, e aplicamos duas regras: Caso a contagem seja maior que o LimVV, com o padrão sendo 4, nós ativamos a célula em questão; caso a contagem seja menor do que o LimVV, também com o padrão sendo 4, nós desativamos a célula em questão. Ao fim dos cinco passos de suavização, nós temos o resultado demonstrado na figura abaixo.

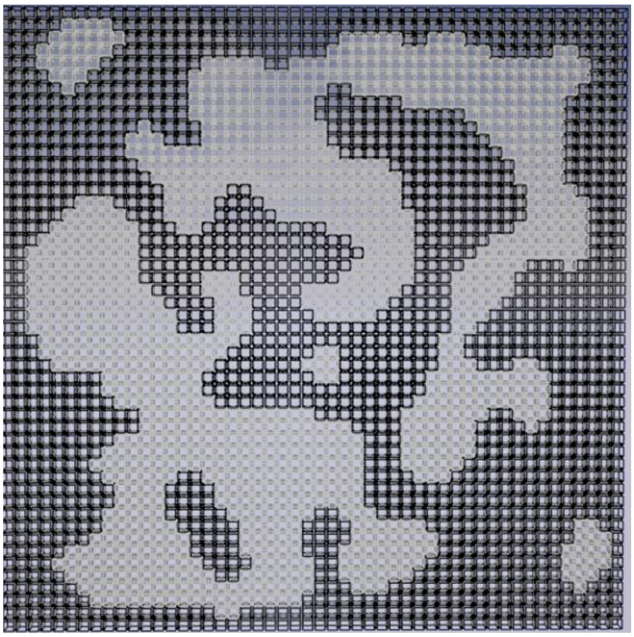

Figura 22. Autômato Celular após 5 passos de suavização. (Elaborado pelo autor)

Com este resultado já é possível notar um contorno bastante interessante para a caverna, porém alguns pontos podem ser levantados, alguns deles sendo:

- A caverna gerada possui duas regiões grandes e três pequenas, caso estas regiões não sejam tratadas de alguma forma o jogador poderá se encontrar em uma das regiões pequenas e não ter acesso ao resto do mapa.
- O contorno em si está composto por quadrados e, a menos que o jogo em questão tenha como design um mundo no estilo 8 bits, este aspecto não é interessante.

Para tratar estas questões, nós daremos continuação a nossa solução, primeiramente atacando a questão das regiões criadas pelo método de suavização.

4.2 Detecção de regiões

Agora que temos o mapa criado, como demonstrado na figura 21, nós iremos detectar as regiões de paredes, células ativas, e de salas, células desativas. Posteriormente, nós iremos destruir as regiões que nós consideramos pequenas de mais. O algoritmo de detecção de regiões de paredes e salas é o mesmo, só mudamos a entrada passada para ser de valor ativo ou desativo. No fim nós iremos aplicar um procedimento de criação de borda para aumentar a espessura da borda e o tamanho do mapa.

O algoritmo é da seguinte forma: Para cada célula no mapa, nós comparamos o valor da célula com o valor de entrada passado, caso os valores sejam iguais, ou seja, se estamos procurando por regiões de salas e encontramos um valor desativo, então nós começamos um procedimento de inundação (*flooding*) para preencher toda a região e guardá-la em uma lista de regiões. Uma vez que detectamos todas as regiões, nós observamos cada região encontrada e comparamos o seu tamanho, a quantidade de células, com a variável "*Room Threshold Size*", com o valor padrão 24, se for uma região de sala, ou "*Wall Threshold Size*", com o valor padrão 8, se for uma região de parede, caso o tamanho seja menor do que o limite, então essa região é destruída, ou seja, transformada no valor oposto. Com isso nós eliminamos regiões muito pequenas, que podem ser não desejadas. Por fim, nós levamos em consideração a variável "*Border Size*", com valor padrão de 1, para criar uma borda de células ativas com a espessura definida na variável. Como resultado nós temos a figura abaixo, podemos observar duas pequenas regiões foram removidas, uma no centro da imagem e outra no canto inferior direito e o mapa agora possui uma borda de uma célula de espessura.

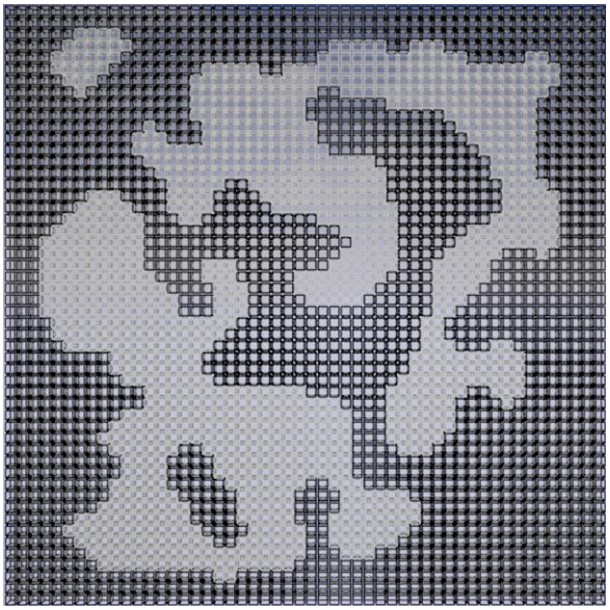

Figura 23. Mapa gerado após detecção de regiões e remoção de regiões pequenas. (Elaborado pelo autor)

Novamente é possível observar que o mapa ainda não está ideal, ambas as questões apresentadas anteriormente não foram tratadas, isto é, o contorno ainda está muito quadrado

e ainda temos regiões separadas umas das outras. Por isso, iremos agora tratar a separação das regiões.

4.3 Conectando as regiões

Nesta etapa nós iremos rodar um procedimento repetidas vezes até que a quantidade de regiões de salas detectadas seja uma, este procedimento irá analisar cada região previamente detectada, encontrar a melhor sala para se criar uma conexão, levando em conta a menor distância possível entre duas salas e então criar a passagem entre as duas salas com um raio de acordo com a variável “*Passage Radius*”, PasR, com valor padrão de 2.

O procedimento a ser repetido irá analisar cada região detectada, de forma que, para cada região o procedimento irá observar todas as outras regiões contidas no mapa, verificando se existe uma conexão entre as duas salas, não havendo conexão ele irá observar todas as células de ambas as salas, calculando a distância entre as duas células e respectivamente entre as duas salas, guardando a melhor distância encontrada juntamente com ambas melhores salas e células. Ao fim da busca de melhor conexão para a sala em questão, o procedimento dará início à criação da passagem. Para a criação da passagem o algoritmo irá traçar uma reta entre as duas melhores células, esta reta terá um raio ou largura do tamanho do PasR, com a reta criada o algoritmo passará por todas as células contidas na reta alterando seu valor para desativado.

Ao fim do procedimento nós temos uma única região de sala, que é composta pelas regiões que existiam anteriormente e foram conectadas com as passagens criadas, como pode ser observado na figura a seguir.

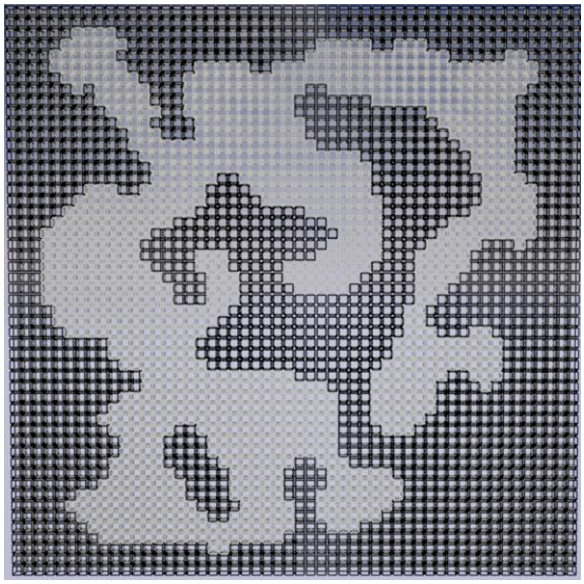

Figura 24. Mapa gerado após conexão das regiões de salas. (Elaborado pelo autor)

Com o resultado da figura acima podemos ver que as três regiões que tínhamos previamente estão agora englobadas em uma única região, resolvendo assim o problema de posicionamento inicial do jogador, personagens e também de objetos. Entretanto, atualmente nós temos apenas quadrados sem colisão e ainda temos o problema de o contorno não ser suave. Iremos então tratar na próxima sessão o problema dos quadrados, os substituindo com *Voxels* 2D.

4.4 *Voxels*

Um *Voxel* é de forma simples uma unidade mínima de volume, geralmente organizada num espaço tridimensional, que guarda uma série de informações [22], no nosso caso cada *Voxel* representa um quadrado e este guarda informações a respeito de cada vizinho e de si mesmo, como se está ativo ou não, sua posição e um inteiro que guarda sua configuração (será mais explicado na sessão de quadrados marchantes). Além desses atributos, o *Voxel* é um objeto 3D com colisão, iluminação, simulação de física, materiais, etc.

Nós utilizaremos *Voxels* por sua versatilidade e a possibilidade de combinação com outras técnicas, como por exemplo os Quadrados Marchantes que serão aplicados na próxima sessão.

Na figura a seguir podemos observar como o nosso mapa seria representado com *Voxels* quadrados com um material que muda sua cor de acordo com a altura.

Figura 25. Mapa gerado com a utilização de Voxels. (Elaborado pelo autor)

Agora que estamos utilizando Voxels nós podemos finalmente atacar o problema de suavização do contorno da caverna, para isso iremos aplicar o algoritmo de Quadrados Marchantes na próxima sessão.

4.5 Quadrados Marchantes

O Algoritmo de Quadrados Marchantes é mais comumente encontrado como Cubos Marchantes, no caso cubos sendo a aplicação em ambientes 3D e quadrados em ambientes 2D, como estamos trabalhando com a geração de um mapa 2D então só iremos falar de Quadrados Marchantes.

O algoritmo em si é bastante simples e elegante, nós temos quatro nós de controle e quatro nós secundários, na figura a seguir podemos ver os nós de controle como os quatro círculos nas extremidades das arestas do quadrado e os nós secundários como os quatro círculos nos centros das arestas dos quadrados.

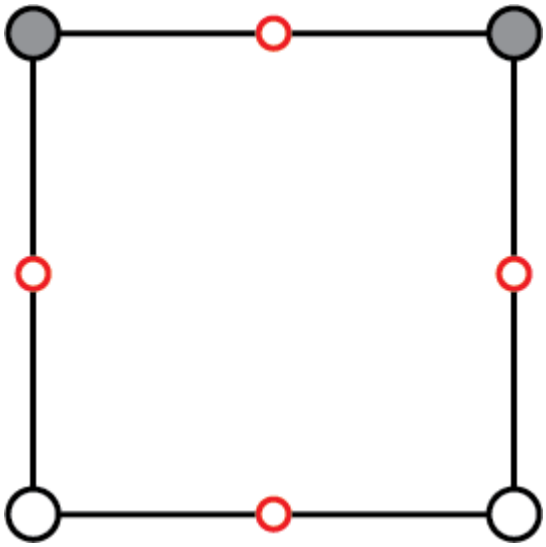

Figura 26. Representação de um Quadrado Marchante. (Elaborado pelo autor)

Cada nó de controle pode estar ativo ou desativo, iremos representar um nó ativo com um círculo preenchido com a cor cinza e um nó desativo com um círculo preenchido com a cor branca. Estes nós de controle serão ativados ou desativados de acordo com a quantidade de células ativas ao redor da célula em questão e conforme estes estados, uma forma é criada dentro dos limites do Quadrado Marchante sendo assim, uma configuração.

No total nós temos dezesseis configurações possíveis para o algoritmo de Quadrados Marchantes, isto já considerando repetições devido a rotações. Na figura a seguir podemos observar estas configurações numeradas de zero a quinze.

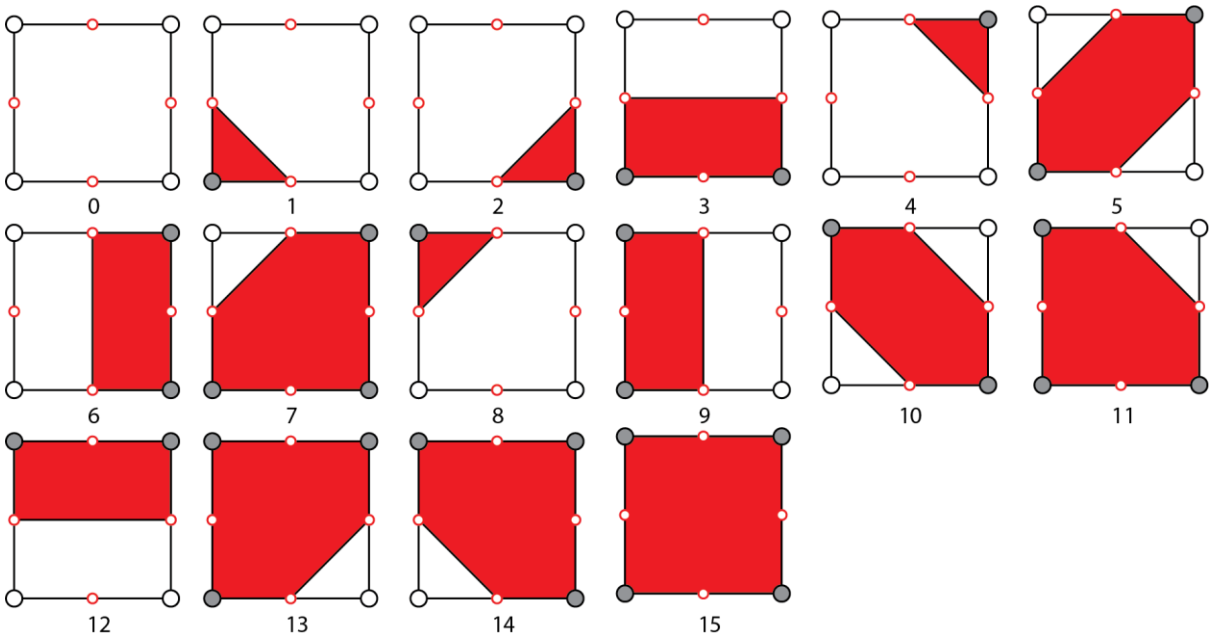

Figura 27. As dezesseis possíveis configurações do algoritmo de Quadrados Marchantes. (Elaborado pelo autor)

Aplicando esta técnica em nossa solução, nós conseguimos suavizar o contorno da caverna, tornando-a mais orgânica, como podemos ver na imagem a seguir.

Figura 28. Mapa gerado com a utilização de Quadrados Marchantes. (Elaborado pelo autor)

Agora, nós temos uma estrutura com um contorno semelhante a uma caverna natural, porém é interessante a criação de um chão ou muro para impedir o jogador de cair ou passar por trás da caverna. A construção deste muro poderia ser feita de forma trivial ao se adicionar profundidade nos *Voxels* existentes, entretanto com essa solução nós mais que duplicaríamos a quantidade de triângulos que precisam ser renderizados para criar a cena. Pensando nessa otimização, nós iremos na próxima sessão detectar as arestas da região e paredes e criar um muro partindo destas arestas.

4.6 Detecção de arestas e criação de muros

Primeiramente nós precisamos pensar que para ser renderizado, cada *Voxel* precisa ser triangularizado de forma a otimizar a quantidade mínima de triângulos necessários para criar cada *Voxel*. Como estamos utilizando Quadrados Marchantes e já sabemos previamente todas as configurações possíveis, nós podemos com facilidade criar os triângulos para cada configuração, na figura abaixo podemos ver como seria esta triangularização para a configuração 14.

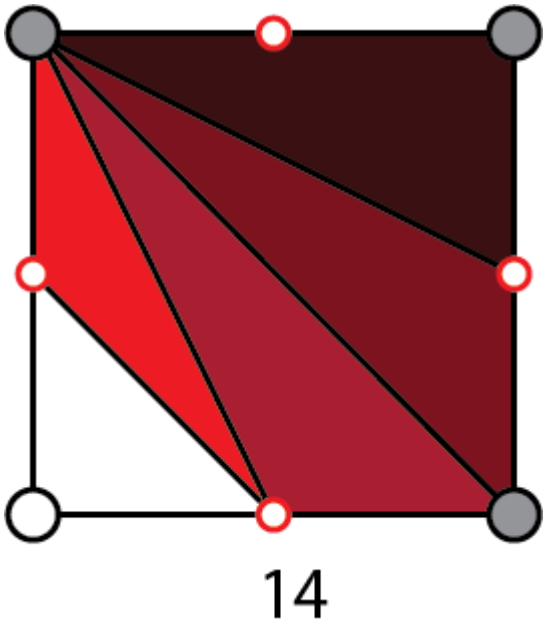

Figura 29. Configuração 14 do Quadrados Marchantes triangularizado. (Elaborado pelo autor)

Agora que temos vários triângulos para um único *Voxel*, nós precisamos uma forma de identifica quando uma dada aresta é uma aresta de borda ou não. Se observarmos atentamente, veremos que uma aresta de borda só pode fazer parte de um triangulo, ou seja, se varrermos todas as arestas dos triângulos criados para gerar o mapa da figura 27, checando a quantidade de triângulos que cada aresta faz parte, nós podemos identificar quais arestas são parte de uma borda ou não.

Uma vez que temos todas as arestas de borda, nós podemos criar dois triângulos para cada aresta, criando então um muro em todo contorno do mapa. Nas seguintes figuras podemos ver o muro criado.

Figura 30. Parte frontal do mapa após a criação do muro. (Elaborado pelo autor)

Figura 31. Parte traseira do mapa após a criação do muro. (Elaborado pelo autor)

Podemos ver que com esta solução nós criamos um muro prático, que irá impedir a passagem do jogador e outros personagens, e ao mesmo tempo otimizado, já que triângulos desnecessários não serão criados.

5 Resultados

Este capítulo está dividido em duas sessões, primeiramente iremos descrever como foi feita nossa implementação, então faremos uma comparação entre os resultados com e sem nossas etapas de processamento, em seguida vamos demonstrar alguns exemplos de mapas gerados com a solução proposta, alterando os parâmetros da solução, com o intuito de mostrar a versatilidade dos resultados. Posteriormente nós iremos falar sobre otimizações que foram e podem ser realizadas.

5.1 Implementação

Para a solução proposta, ao invés de partirmos de uma linguagem de programação qualquer, utilizando bibliotecas como DirectX ou OpenGL, para desenvolvermos a solução, nós decidimos utilizar uma *Engine*, ou motor gráfico, no caso a Unreal Engine 4, para facilitar e acelerar este desenvolvimento, ao mesmo tempo que testávamos as capacidades e limitações de seu sistema de *Visual Scripting,* chamado *Blueprints*. Na sessão a seguir iremos detalhar o que são *Engines*, a Unreal Engine 4 e seu sistema *Blueprints*.

5.1.1 Unreal Engine

No desenvolvimento de jogos, o desenvolvedor se encontra em um processo de utilizar múltiplas estruturas do computador para diversas finalidades, como ler as entradas dos jogadores, simular física e colisões, criar objetos 2D e 3D, emitir sons e muito mais. Para isso, múltiplas bibliotecas são utilizadas ou criadas especificamente para o jogo, alguns exemplos são a utilização do DirectX ou OpenGL para a reprodução de imagem e áudio para o jogador. Comumente essas bibliotecas são compiladas em conjunto para facilitar sua utilização e agilizar possíveis futuros trabalhos, eventualmente uma dessas compilações engloba tantos aspectos do desenvolvimento que passa a ser chamada de *Game Engine*, ou motores gráficos [7]. Alguns grandes exemplos de motores gráficos são Unity, Unreal Engine, CryEngine,

Frostbite e outras. É importante notar que muitas dessas *Engines* foram criadas durante o processo de desenvolvimento de um jogo especifico e por isso acabam sendo “especialistas” em certas áreas.

Nós decidimos utilizar a Unreal Engine 4, UE4, desenvolvida pela Epic Games durante a produção do jogo Unreal Tournament, pelos benefícios da utilização de um motor gráfico e por alguns motivos específicos da UE4, como: É completamente grátis para atividades não lucrativas, *Open Source*, por conhecimento prévio e pela curiosidade de descobrir o que pode ser alcançado com a UE4 e seu sistema de *Visual Scripting* próprio, chamado *Blueprints*, para criação de conteúdo procedimental.

Na figura abaixo vemos um exemplo de *Blueprints* com as etapas da solução proposta sendo chamadas no evento de *Tick*, que é chamado a cada quadro do jogo.

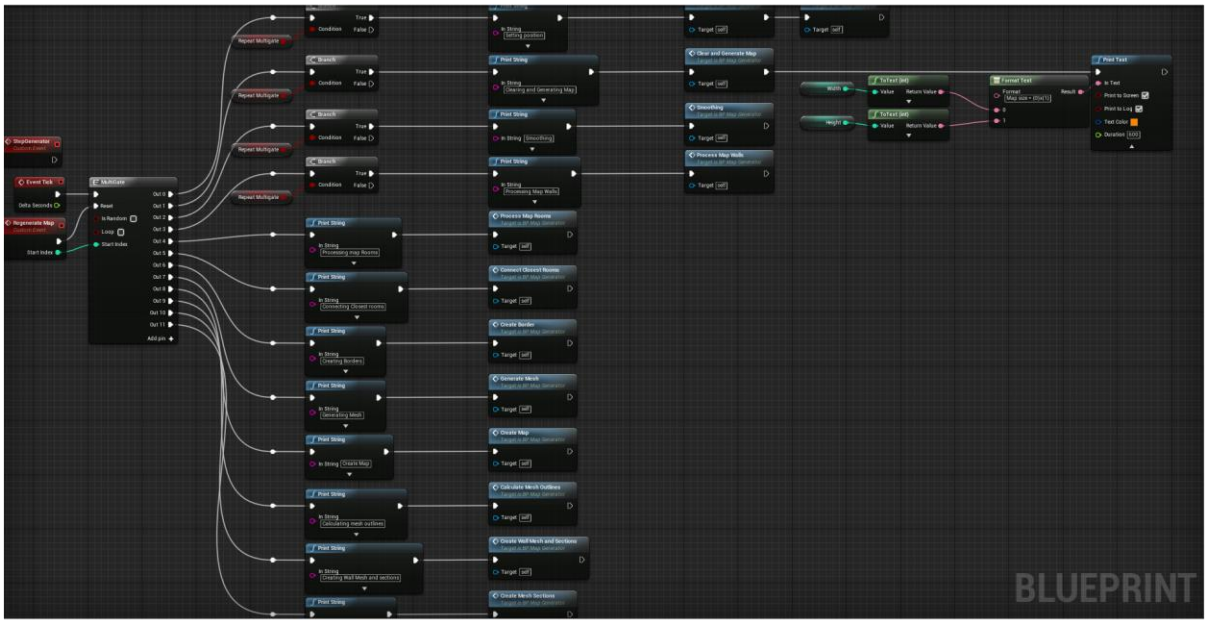

Figura 32. Etapas da solução proposta em Blueprints (Elaborado pelo autor)

A ideia principal por trás de Blueprints é acelerar o desenvolvimento, ao mesmo tempo que permite pessoas que não sejam programadoras se aventurarem no desenvolvimento de jogos.

5.2 Autômato Celular vs Solução Proposta

Para efeito de comparação, nós iremos apresentar a seguir uma série de figuras de cavernas geradas, de um lado teremos a caverna gerada com o Autômato Celular puro e do outro a caverna gerada com a nossa solução, conforme descrevemos no capítulo 4, ambas com a mesma semente.

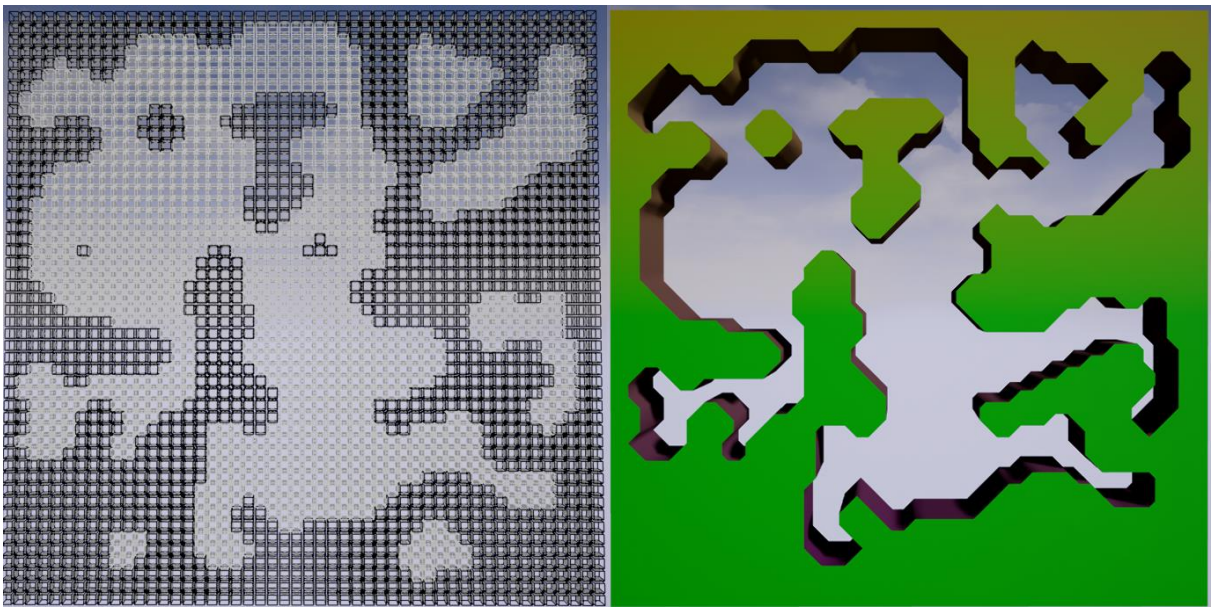

Figura 33. Resultado do Autômato Celular e da solução proposta para a semente 140189. (Elaborado pelo autor)

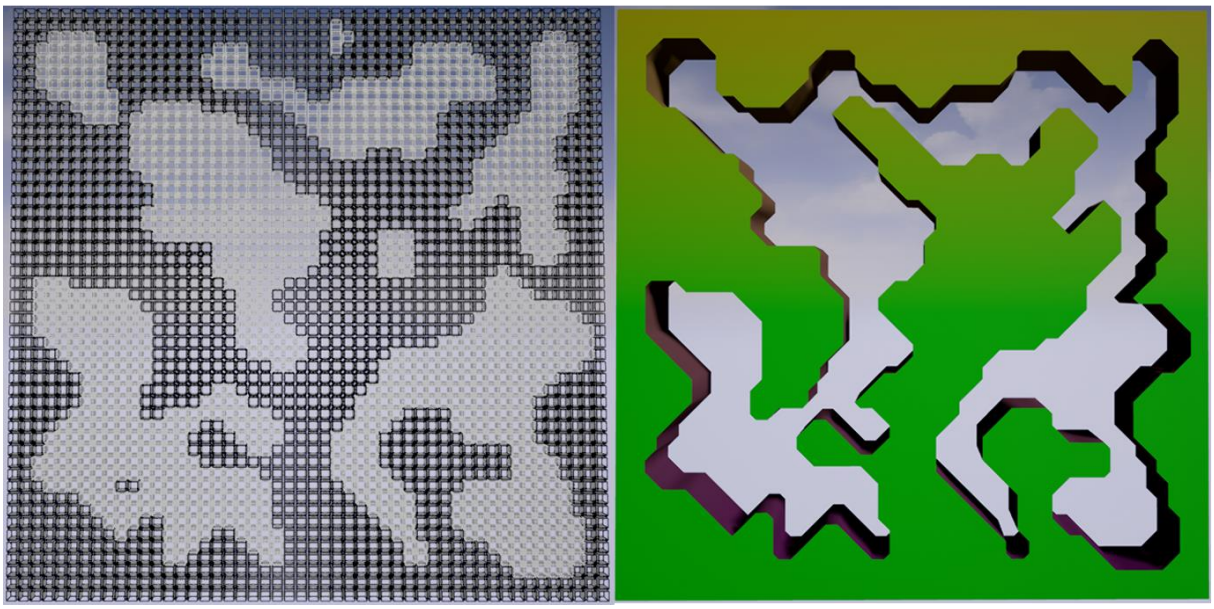

Figura 34. Resultado do Autômato Celular e da solução proposta para a semente 150992. (Elaborado pelo autor)

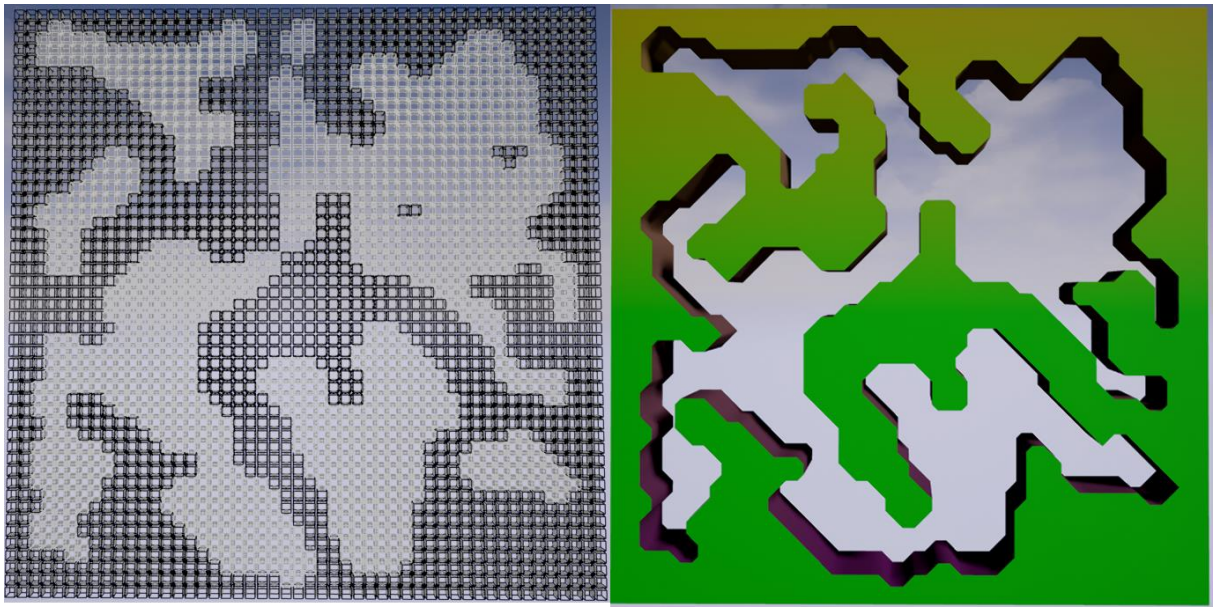

Figura 35. Resultado do Autômato Celular e da solução proposta para a semente 7092014. (Elaborado pelo autor)

Podemos observer em cada uma das três figuras acima a utilidade das etapas citadas no capítulo 4, no caso regiões pequenas de mais, tanto de sala como de paredes foram removidas, todas as salas foram conectadas, o contorno foi todo suavizado e foi criado um muro nos limites da caverna para limitar a movimentação do jogador.

5.3 Mapas e parâmetros

Como dito no decorrer deste trabalho, a utilização de uma semente permite alcançar o mesmo mapa todas as vezes, porém, isto ocorre ao manter os outros parâmetros do algoritmo fixo, se por exemplo alterarmos o tamanho do mapa ou a PbVI, o mapa gerado será completamente diferente. No caso, ao mantermos a semente de teste em 842498 e mudarmos estes dois parâmetros, considerando três configurações de exemplo, a primeira sendo o mapa original, a segunda o mapa com tamanho 75x75 *Voxels* e a terceira o mapa com tamanho 50x50 *Voxels* e PbVI 45 ao invés de 50, nós teremos os seguintes resultados, da esquerda para a direita respectivamente.

Figura 36. Mapa na configuração 1 com a semente 842498. (Elaborado pelo autor)

Figura 37. Mapa na configuração 2 com a semente 842498. (Elaborado pelo autor)

Figura 38. Mapa na configuração 3 com a semente 842498. (Elaborado pelo autor)

É fácil perceber que pequenas alterações podem resultar em mapas completamente diferentes, de forma que nós temos uma infinidade de possibilidades para cada semente, como demonstrado com a semente 842498. Alguns parâmetros que são facilmente manipulados são: O comprimento e largura do mapa; A PbVI; A semente; as dimensões dos *Voxels*; a quantidade de passos de suavização do algoritmo de Autômato Celular; A largura da borda adicionada no passo 4.2; a largura das passagens; o LimVM e vivos; o limite do tamanho de muros e salas. Todos estes parâmetros causam mudanças significantes no contorno da caverna a ser gerada.

5.4 Otimizações

Durante a implementação do projeto a criação dos triângulos dos *Voxels* estava sendo feita de forma trivial, de forma que em um determinado mapa de tamanho 15x15 *Voxels* estavam sendo criados cerca de 900 vértices, foi então implementado um algoritmo simples para impedir a sobreposição desses vértices e o número total desceu para cerca de 270.
Outro ponto claro de otimização é a própria utilização de Blueprints, esta pode ser trocada por C++. Os próprios desenvolvedores da UE4 afirmam que em certos cenários o mesmo código em C++ pode ser até dez vezes mais rápido do que em Blueprints. Este desempenho é notado com mapas grandes, no caso mapas maiores do que 100x100 *Voxels*, já levam um tempo médio considerável para ser gerado. No caso o mapa abaixo foi gerado em cerca de um minuto e vinte segundos, contra cerca de sete segundos para a criação do mapa teste da figura 31.

Figura 39. Mapa 100x100 Voxels. (Elaborado pelo autor)

É preciso notar que um dos objetivos deste trabalho era analisar as limitações das Blueprints, por isso a não utilização de C++, outro ponto é o fato de desempenho não ser um objetivo deste trabalho, as otimizações implementadas durante a implementação foram realizadas para agilizar o processo de desenvolvimento e testes e não para alcançar um bom desempenho geral.

6 Conclusão

O trabalho presente apresentou um contexto do mundo de desenvolvimento de jogos, mostrando sua relevância em meio a outras indústrias de entretenimento, e como alguns jogos conseguem rivalizar grandes sucessos de *Hollywood*. Revelamos também a importância de GPC na indústria de jogos, exemplificando a sua versatilidade em vários jogos, com a geração de armas, criaturas, sons e muito mais. Mostramos também a utilização de GPC em criação de mapas juntamente com técnicas comuns na criação de cavernas, detalhando estas técnicas e exemplificando suas utilizações. Então, partimos para a solução proposta, implementamos um sistema robusto e simples que pode ser utilizado para gerar uma infinidade de cavernas 2D, com aspectos semelhantes ao de cavernas naturais.

Nós pudemos então com esta solução alcançar o nosso objetivo geral, ou seja, criamos um gerador de cavernas 2D com contornos semelhantes ao de cavernas naturais, utilizando as seguintes seis etapas detalhadas no quarto capítulo, que podem ser resumidas na utilização de Autômatos Celulares para a criação do contorno da caverna e então o processamento em cima da caverna gerada para garantir acesso a todas as regiões criadas e a suavização do contorno com *Voxels* e Quadrados Marchantes. Sendo assim, com nossa solução é possível acelerar o processo de desenvolvimento de jogos, já que uma caverna que poderia demorar dias para ser planejada e criada pode ser gerada em poucos segundos. Atendemos também nossos objetivos específicos, isto é, exploramos algumas técnicas que são comumente utilizadas no mercado.

Outra contribuição mencionável é a implementação propriamente dita, como explicado no capítulo anterior, nós pudemos avaliar a solução proposta em um ambiente real de desenvolvimento de jogos, a Unreal Engine 4, com a utilização de seu sistema de *Visual Scripting*, *Blueprints*, demonstrado sua capacidade e detectando alguns problemas e dificuldades como detalharemos na próxima sessão.

6.1 Dificuldades

A maior dificuldade encontrada durante o desenvolvimento de nossa solução foi a utilização do sistema Blueprints, é um sistema que foi criado para a UE4 e ainda está em processo de desenvolvimento, por isso, além do fato do desempenho ser reduzido, algumas práticas de desenvolvimento não podem ser adotadas e outras estão altamente limitadas. Alguns exemplos são por exemplo a criação de *Arrays* multidimensionais, alguns artifícios precisam ser criados para alcançar o mesmo resultado esperado, por exemplo criar um *Array* de apenas uma dimensão e fazer o acesso através de uma formula matemática com o tamanho da coluna, ou a utilização de *Structs* para simular uma coluna e então fazer um *Array* unidimensional de Structs de colunas.

Outro exemplo é a quantidade de ações em cada quadro, no caso o sistema de Blueprints apresenta atualmente uma limitação para a quantidade máxima de ações que podem ocorrer em cada quadro atualizado do jogo (*Tick*), este limite tem um padrão de um milhão de ações e pode ser aumentado consideravelmente, mas existe um limite para o valor a ser inserido. Este limite de um milhão de ações pode parecer um grande valor, porém uma simples estrutura de repetição de um *For*, por exemplo executa quatro ações a cada iteração, ou seja, se formos percorrer cada posição de um mapa de 50x50 *Voxels*, nós temos um total de $50x50x4$ ações, totalizando dez mil ações, isto sem contar as ações tomadas a cada iteração. Podemos ver então que trabalhando com mapas esse valor limite seria alcançado rapidamente e foi alcançado, mas como demonstrado na figura 1, há uma forma de contornar este problema, nós separamos as etapas do algoritmo de forma que só executamos uma pequena parte da solução a cada quadro.

Além dos problemas já citado, o sistema de Blueprints possuem um limite também para a quantidade de chamadas numa função recursiva de 250 chamadas e isto impediu a utilização de recursão no procedimento de detecção de arestas, por exemplo.

6.2 Trabalhos futuros

Pensamos em alguns projetos futuros que podem ser realizados com a utilização desta solução e alguns que mais chamam a atenção são: A conversão da solução para C++, para observar o aumento de desempenho e evitar as limitações citadas a cima; A implementação de uma interface para a fácil utilização do sistema; A possível criação de um *Plugin* para ser lançado no mercado de recursos da UE4; A adaptação e utilização da solução em um jogo; A adaptação da solução para um sistema de mapas infinitos gerados em tempo real.

Referências

[1] COX, Kate. **It's Time To Start Treating Video Game Industry Like The $21 Billion Business It Is.** 2014. Disponível em: <https://consumerist.com/2014/06/09/its-time-to-start-treating-video-game-industry-like-the-21-billion-business-it-is/>. Acesso em: 11/04/2016

[2] TAKATSUKI, Yo. **Cost headache for game developers.** 2007. Disponível em: <http://news.bbc.co.uk/2/hi/business/7151961.stm>. Acesso em: 30/03/2016

[3] _____. **Procedural Content Generation: Thinking With Modules.** 2012. Disponível em: <http://www.gamasutra.com/view/feature/174311/procedural_content_generation_.php>. Acesso em: 11/04/2016

[4] TOGELIUS, Julian, et al. **What is Procedural Content Generation? Mario on the borderline**. Proceedings of the Foundations of Digital Games Conference. 2011

[5] LUDWIG, Joe. **Procedural content is hard**. 2007. Disponível em: <http://programmerjoe.com/2007/02/11/procedural-content-is-hard/>. Acesso em: 30/03/2016

[6] LEE, Joel. **No Man's Sky and the Future of Procedural Games**. 2015. Disponível em: <http://www.makeuseof.com/tag/no-mans-sky-future-procedural-games/>. Acesso em: 11/04/2016

[7] ENGER, Michael. **Game Engines: How do they work?**. 2013. Disponível em: <http://www.giantbomb.com/profile/michaelenger/blog/game-engines-how-do-they-work/101529/>. Acesso em: 14/04/2016

[8] HENDRIKX, Mark, et al. **Procedural Content Generation for Games: A Survey**. Multimedia Computing Communications and Applications, Artigo 1. 2011.

[9] DE CARLI, Daniel, et al. **A survey of procedural content generation techniques suitable to game development**. Brazilian Symposium on Games and Digital Enternainment(SBGAMES). 2011.

[10] LABSCHÜTZ, Matthias, et al. **Content Creation for a 3D Game with Maya and Unity 3D**.2011.

[11] BOYER, Brandon. **My generation: How Indie Game Makers are Embracing Controlled Chaos**. 2009. Disponível em: <http://boingboing.net/2009/10/12/my-generation-how-in.html>. Acesso em: 10/04/2016.

[12] FINGAS, Jon. **Here's how 'Minecraft' creates its gigantic worlds**. 2015. Disponivel em: <https://www.engadget.com/2015/03/04/how-minecraft-worlds-are-made/. Acesso em 02/07/2016>.

[13] **Borderlands Wikia**. Disponivel em: <http://borderlands.wikia.com/wiki/Weapons>. Acesso em: 02/07/2016

[14] MCMILLEN, Edmund. **The Binding of Isaac Gameplay explained**. 2011. Disponível em: <http://edmundmcmillen.blogspot.com.br/2011/09/binding-of-isaac-gameplay-explained.html>. Acesso em: 03/07/2016.

[15] WILLIAMS, Nathan. **An Investigation in Techniques used to Procedurally Generate Dungeon Structures**. 2014.

[16] FUCHS, Henry, et al. **On visible surface generation by a priori tree structures**. SIGGRAPH '80 Proceedings of the 7th annual conference on Computer graphics and interactive techniques. 1980.

[17] **Basic BSP Dungeon Generation**. 2012. Disponível em: <http://www.roguebasin.com/index.php?title=Basic_BSP_Dungeon_generation>. Acesso em: 04/07/2016.

[18] DELAUNAY, Boris. ***Sur la sphère vide, Izvestia Akademii Nauk SSSR, Otdelenie Matematicheskikh i Estestvennykh Nauk****, 7:793–800, 1934.*

[19] ADONAAC, A. **Procedural Dungeon Generation Algorithm**. 2015. Disponível em: <http://www.gamasutra.com/blogs/AAdonaac/20150903/252889/Procedural_Dungeon_Generation_Algorithm.php>. Acesso em: 08/07/2016.

[20] JOHNSON, Lawrence, et al. **Cellular automata for real-time generation of infinite cave levels**. 2010.

[21] COOK, Michael. **Generate Random Cave Levels Using Cellular Automata**. 2013. Disponível em: <http://gamedevelopment.tutsplus.com/tutorials/generate-random-cave-levels-using-cellular-automata--gamedev-9664>. Acesso em: 10/04/2016.

[22] SANTAMARÍA-IBIRIKA, Aitor, et al. **Volumetric Virtual Worlds with Layered Terrain Generation**. International Conference on Cyberworlds. 2013.

[23] LAGUE, Sebastian. **[Unity] Procedural Cave Generation**. 2015. Disponível em: <https://www.youtube.com/watch?v=v7yyZZjF1z4>. Acesso em 11/04/2016.

[24] _____. **Performance comparison between Blueprints and C++?**. 2014. Disponível em <https://answers.unrealengine.com/questions/41895/performance-comparison-between-blueprints-c.html>. Acesso em: 11/04/2016.

[25] _____. **Minecraft 1.7 Biome Changes.** 2013. Disponível em: <http://www.planetminecraft.com/blog/thinking-over-minecraft-17-biome-changes/>. Acesso em 07/04/2016.

[26] YEGGE, Steve. **The Borderlands Gun Collector's Club.** 2012**.** Disponível em: <http://steve-yegge.blogspot.com.br/2012/03/borderlands-gun-collectors-club.html>. Acesso em 07/04/2016.

[27] _____. **Need Help with Binding of Isaac style UMG Mapscreen**. 2015. Disponível em: <https://answers.unrealengine.com/questions/193941/need-help-with-binding-of-isaac-style-umg-mapscree.html>. Acesso em 07/04/2016.

[28] _____. **Rooms**. 2016. Disponível em: <http://bindingofisaacrebirth.gamepedia.com/Rooms>. Acesso em 07/04/2016.

[29] _____. **Spore Creature Creator for Mac**. 2014. Disponível em: <http://mac.softpedia.com/get/Games/Spore-Creature-Creator.shtml>. Acesso em 07/04/2016.

[30] _____. **Development of Spore.** 2015. Disponível em: <http://spore.wikia.com/wiki/Development_of_Spore>. Acesso em 07/05/2016.

[31] _____. **Gypsum Cave Album**. 1933.Disponível em: <http://www.nevadarockart.info/photo-gallery-images/gypsumcave/album/index.html>. Acesso em 17/07/2016.