Um Método Automático de Geração de Terrenos 3D para Jogos *Low-poly*

Jam Sávio Ferreira da Conceição [1], Jadson Lúcio dos Santos[1], Rodolfo C. Cavalcante [1]

[1]*Núcleo de Ciências Exatas (NCEx)*
Universidade Federal de Alagoas
Arapiraca, Alagoas

{jam.conceicao,jadson.santos,rodolfo.cavalcante}@arapiraca.ufal.br

Abstract. *The creation of 3D maps plays a fundamental role in the process of building a game. This step usually takes a considerable time to complete, since adjustments in the land format are done manually by a graphic designer who loses considerable time defining the best format and structure for the created map. In this paper we present a method for automatic generation of 3D low-poly terrain from images, which although not of general purpose, can help the graphic designer in the rapid development of the land structure through drawings, which unlike the iterative manual process, can be done quickly.*

Resumo. *A criação de mapas 3D tem um papel fundamental no processo de construção de um jogo. Essa etapa costuma levar um tempo considerável para ser executada, pois ajustes no formato do terreno são feitos manualmente pelo designer gráfico, que perde um tempo considerável definindo o melhor formato e estrutura para o mapa criado. Nesse trabalho, nós apresentamos um método para geração automática de terrenos 3D* low-poly *a partir de imagens, que embora não sejam de propósito geral, pode auxiliar o designer gráfico no desenvolvimento da estrutura de um terreno através de desenhos. A principal contribuição desse trabalho é um método que acelera o processo de construção de mapas 3D.*

1. Introdução

A indústria dos jogos tem se mostrado uma indústria forte e lucrativa. Apenas em 2019, ela faturou cerca de 159 bilhões de dólares [1], arrecadando mais dinheiro que a indústria de cinema e música juntas, que somaram cerca de 60 bilhões de dólares em 2018 [2]. Dentro desse contexto, os jogos *low-poly* tem ganhado cada vez mais atenção, com títulos como *Morphite* [3] e *Fruits of a Feather* [4]. *Low-poly* é um termo utilizado para designar objetos ou mapas 3D que possuem uma baixa quantidade de polígonos, assim como uma representação de cores que deixe em evidência esses polígonos. A Figura 1 mostra um exemplo desse tipo de mapa.

[1]https://newzoo.com/insights/articles/newzoo-games-market-numbers-revenues-and-audience-2020-2023/

[2]https://www.ejinsight.com/eji/article/id/2280405/20191022-video-game-industry-silently-taking-over-entertainment-world

[3]https://store.steampowered.com/app/661740/Morphite/

[4]https://samuraipunk.itch.io/feather

Figura 1. Exemplo de mapa low-poly

Fonte: Página de rede social Pinteres [5]

Embora haja um crescimento acentuado da indústria de jogos nos últimos anos [Marchand and Hennig-Thurau 2013], o custo médio para se criar um jogo também aumentou [6]. Um dos motivos para isso é que o processo para criação de um jogo tornou-se mais sofisticado, com equipes de desenvolvimento cada vez maiores. Nesse contexto, o designer é responsável pela criação dos mapas. Porém, devido ao seu caráter manual inerente, quando esse processo é realizado em um jogo com grandes mapas, o tempo que o designer gasta para desenvolver a base 3D de um mapa pode aumentar consideravelmente o cronograma de desenvolvimento, o que acaba elevando os custos de produção do jogo [De Carli et al. 2011].

Diante disso, este artigo propõe um método automático para geração de terrenos 3D no estilo *low-poly* a partir de imagens desenhadas pelo usuário. Nós acreditamos que esse método pode acelerar o processo de desenvolvimento dos mapas dos jogos, uma vez que o designer pode simplesmente desenhar em 2D o modelo do mapa a ser projetado e o método proposto produz, em tempo real, o modelo 3D correspondente, sem a necessidade de ficar lidando diretamente com ângulos de visão e ajustes diretos do modelo 3D em algum software de modelagem.

O restante desse artigo está organizado da seguinte forma. Na Seção 2 são discutidos alguns trabalhos relacionados a esta pesquisa. Na Seção 3 é descrita a abordagem proposta e as ferramentas usadas para implementá-la. A Seção 4 conclui o artigo com aplicações para a abordagem proposta e algumas formas de como essa pesquisa pode ser continuada em trabalhos futuros.

2. Trabalhos relacionados

A geração dos mapas é uma parte fundamental do processo de criação de um jogo, pois a qualidade desse tipo de conteúdo impacta diretamente no custo do projeto [De Carli et al. 2011]. Essa tarefa geralmente requer uma equipe composta por vários artistas e modeladores 3D [De Carli et al. 2011]. Diante disto, vários trabalhos na literatura buscam técnicas para facilitar a geração de mapas 3D. Dentre as principais técnicas propostas na literatura temos a geração procedural [De Carli et al. 2011], que busca gerar os mapas aleatoriamente a partir de algum algoritmo predefinido e a geração de mapas a

[6] https://venturebeat.com/2018/01/23/the-cost-of-games/

partir artefatos preexistentes como imagens realísticas dos terrenos [Hung et al. 1998] ou dados geológicos [Parberry 2014].

Uma vantagem da técnica de geração procedural é que ela pode gerar mapas gigantes sem ser necessário gastar tempo modelando-os. No entanto, caso o algoritmo não seja bem configurado, a geração do mapa pode não atender às expectativas desejadas [Amato 2017]. Esse é um dos principais motivos pelo qual a geração procedural não é adotada por um número maior de desenvolvedores. A geração a partir artefatos preexistentes, por outro lado, geralmente possui uma diversidade maior, mas em contrapartida gera terrenos mais suscetíveis a ruídos.

Muitos dos designer gráficos preferem construir o mapa completamente de forma manual [Amato 2017] a partir de ferramentas como o Blender [Brito 2007], pois eles têm um maior controle sobre o resultado final. Porém, esse tipo de abordagem pode levar um tempo significativamente maior, o que acarreta em maiores custos de produção do jogo. Nesse contexto, este artigo apresenta uma técnica para geração de terrenos 3D a partir de imagens 2D que pode acelerar o processo de criação do formato e da estrutura principal do terreno. Esse mapa produzido precisará apenas de pequenas modificações por parte do designer para dar acabamento final ao terreno.

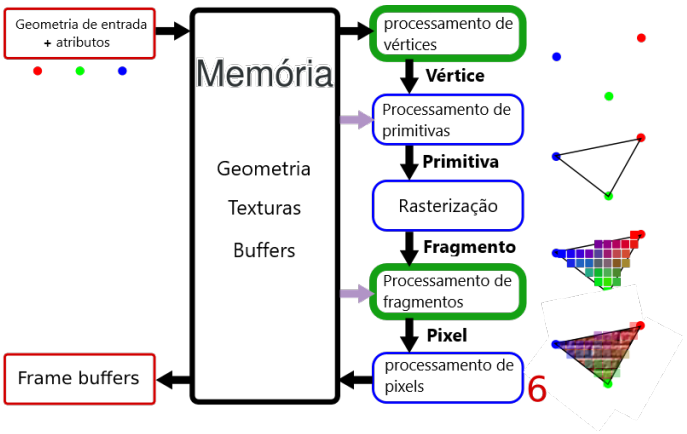
3. Método

Para gerar uma cena 3D as implementações das APIs gráficas utilizam um processo chamado *pipeline* gráfico [Angel et al. 2012]. Este *pipeline* pode variar de uma API gráfica para outra, mas ele sempre segue o conceito de divisão do trabalho em uma sequência de partes menores, em que cada parte é responsável por realizar uma tarefa específica da geração de cena. No final de cada tarefa, o seu resultado é passado para a próxima etapa do *pipeline*. O OpenGL [Shreiner et al. 2013] foi a API gráfica escolhida para gerar os terrenos 3D nesse trabalho e, para a implementação foi utilizada a linguagem de programação Python [Lutz 2001], usando uma biblioteca conhecida como PyOpenGL [Fletcher and Liebscher 2005].

A Figura 2 mostra o *pipeline* gráfico utilizado pelo OpenGL. Esse *pipeline* permite criar um *frame buffer*, que é uma matriz em que cada elemento representa um *pixel*, que posteriormente vai ser mostrado na tela do dispositivo de saída. A primeira etapa é o processamento de vértices. Nessa etapa, as coordenadas originais o que formam os triângulos que vão compor os objetos da cena, e que estão salvas no *vertex buffer*, são transformadas em coordenadas do mundo através de uma série de multiplicações de matrizes. Essas coordenadas transformadas são passadas para a próxima etapa do *pipeline*.

Na segunda etapa essas coordenadas, com o auxílio do *index buffer*, que contém uma lista que informa qual vértice está ligado com qual, vão ser interligadas para formar primitivas geométricas, como triângulos e quadriláteros. Essas primitivas dão o formato dos objetos da cena e são usadas na próxima etapa do *pipeline*, que é conhecida como rasterização. O papel da rasterização é pegar as primitivas gráficas e a cor de cada vértice, que geralmente está armazenado no *vertex buffer*, e fazer uma interpolação dessas cores para descobrir a cor dos *pixels* entre os vértices das primitivas gráficas. O resultado desse processo é a geração da primeira versão do *frame buffer*. As duas próximas etapas vão trabalhar na versão inacabada do *frame buffer* gerado pela etapa anterior, aplicando efeitos de iluminação e texturas nos objetos.

Figura 2. *Pipeline* gráfico do OpenGL



Fonte: Imagem editada de Romain Vergne [7]

Este artigo investiga o processo de geração do *vertex buffer* com as coordenadas e as cores de cada vértice, e também do *index buffer*, que vai conectar esses vértices. Esses dados são utilizados pelo *pipeline* do OpenGL para gerar a cena 3D do terreno. Esse processo precisa inicialmente de uma imagem 2D, que representa o mapa a ser gerado (como pode ser visto na Figura 3), uma tabela com a paleta de cores da imagem e a sua respectiva altura (Figura 4), que vai ser usado para determinar o valor da altura dos vértices na cena que representa o terreno. A altura de uma cor na paleta de cores escolhidas pode variar no intervalo $[-1, 1]$, em que 1 significa que a cor representa elevação máxima e -1 depressão máxima. A imagem do mapa 2D pode ser desenhada pelo designer em um processo de criação de terreno iterativo em tempo real ou gerada por algum algoritmo ou método, como as GANs (*generative adversarial networks*) [Radford et al. 2015].

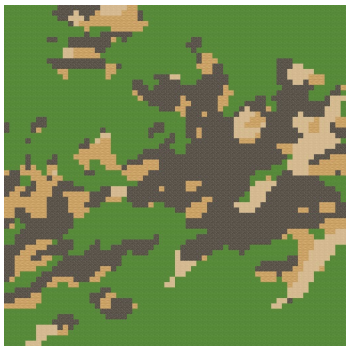


Figura 3. Imagem do mapa que vai ser criado

Cor(RGB)	Tipo terreno	Altura
(121, 183, 220)	Água	-0.05
(171, 201, 175)	Grama	0.0
(238, 240, 191)	Areia	0.0
(213, 215, 214)	Rochas	0.05
(218, 199, 189)	Dunas	0.025

Figura 4. Paleta de cores e as suas respectivas alturas

[7] http://romain.vergne.free.fr/teaching/IS/SI03-pipeline.html

O algoritmo proposto utiliza essas entradas para gerar um *grid* de vértices com a resolução da imagem 2D, como mostrado na Figura 5, que representa uma parte do *grid* gerado pela imagem na Figura 3. Inicialmente todos os vértices desse *grid* vão estar com a posição y das suas coordenadas zeradas, o que gera um mapa sem relevo. O Algoritmo 1 é responsável por gerar a lista com as coordenadas dos vértices e a lista com a cor de cada vértice.

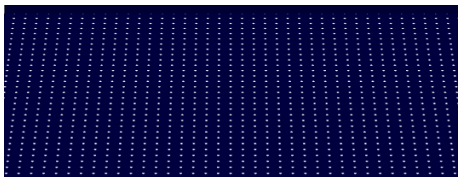


Figura 5. *grid* gerado

Algorithm 1 Algoritmo para geração dos vértices e das cores

$imagem$ é a matrix de *pixels* (r,g,b) da imagem
$vertices$ é a lista de coordenadas dos vértices
$colors$ é a lista de cores de cada vértice
$width$ é a largura da imagem
$height$ é a altura da imagem
for i in $0 : width$ **do**
  for j in $0 : heigh$ **do**
    $vertices \leftarrow vertices + [i, 0, j]$
    $colors \leftarrow colors + imagem[i][j]$
  end for
end for

O próximo passo é gerar o *index buffer* desse mapa, que vai conectar os vértices do mapa e formar as primitivas gráficas. Nesse caso o Algoritmo 2 é responsável por criar esse *index buffer* e gerar as primitivas gráficas, que nesse caso são triângulos. O *index buffer* guarda apenas os índices dos vértices, que estão ordenados da esquerda para a direita e de cima para baixo como mostrado na Figura 6. A Figura 7 mostra o terreno após a execução do Algoritmo 2.

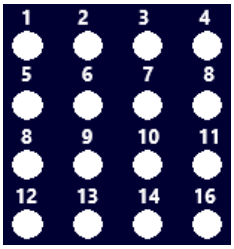


Figura 6. Índices de um *grid* 4x4

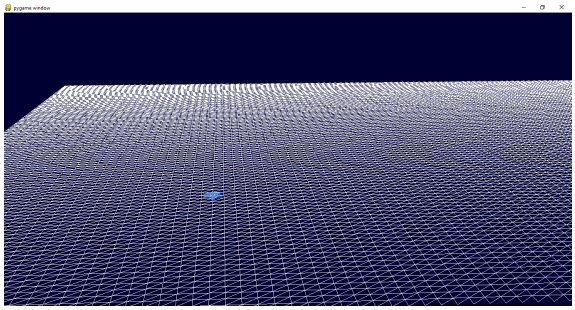


Figura 7. *grid* do mapa com as primitivas

Algorithm 2 Algoritmo para geração do *index buffer* do mapa

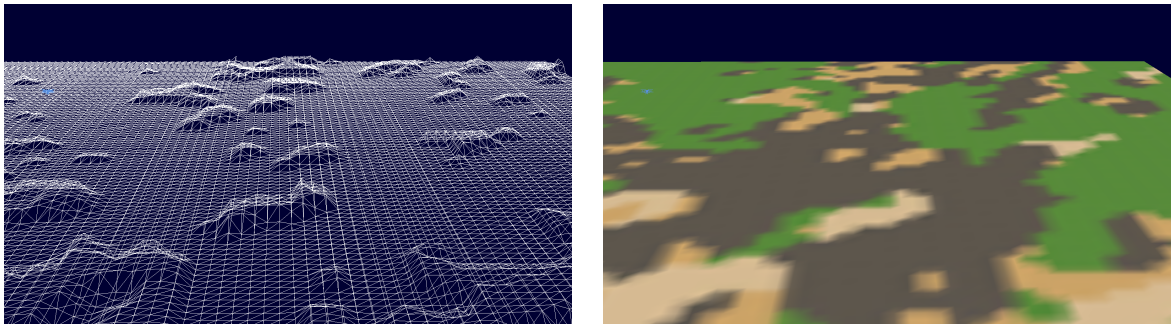
ind_buffer é a lista de índices
$colors$ é a lista de cores dos vértices do *grid*
$width$ é a largura do *grid*
$height$ é a altura do *grid*
for i in $0 : width - 1$ **do**
  for j in $0 : height - 1$ **do**
    $fist\_point \leftarrow j + i * width$
    $ind\_buffer \leftarrow ind\_buffer + [fist\_point, fist\_point + 1, fist\_point + width]$
    $ind\_buffer \leftarrow ind\_buffer + [fist\_point + 1, fist\_point + width, fist\_point + width + 1]$
  end for
end for

Até o momento, os vértices estão com o valor da coordenada y, ou seja, a altura, nulo. A Equação 1 mostra o cálculo dessa altura para o vértice (x, y) no *grid*, onde N é o número de cores na paleta, $altura(c)$ é o valor da altura da cor c que foi previamente definido pelo usuário, a $cor\_pixel(x, y)$ é a cor (r,g,b) do *pixel* na posição (x,y) da imagem que vai gerar o mapa, que também é definido pelo usuário, como mostra a Figura 3. Já a $cor(c)$ é o valor (r, g, b) da cor de índice c na paleta de cores. Por fim, a Equação 2 mostra a distância euclidiana entre a cor do *pixel* (x, y) da imagem e a cor de índice c da paleta de cores, definidos respectivamente como $(r1, g1, b1)$ e $(r2, g2, b2)$.

$$altura(x, y) = \frac{\sum_{c=1}^{N} \frac{altura(c)}{euclid\_dist(cor\_pixel(x,y), cor(c))}}{\sum_{c=1}^{N} \frac{1}{euclid\_dist(cor\_pixel(x,y), cor(c))}} \quad (1)$$

$$euclid\_dist((r1, g1, b1), (r2, g2, b2)) = \sqrt{(r1 - r2)^2 + (g1 - g2)^2 + (b1 - b2)^2} \quad (2)$$

Essa Equação calcula a média ponderada entre as cores disponíveis na paleta e a sua proximidade com as cores da imagem do terreno. A Figura 8 mostra o resultado do cálculo da altura para cada vértice do *grid* para a imagem mostrada na Figura 3 e a paleta definida na Figura 4.

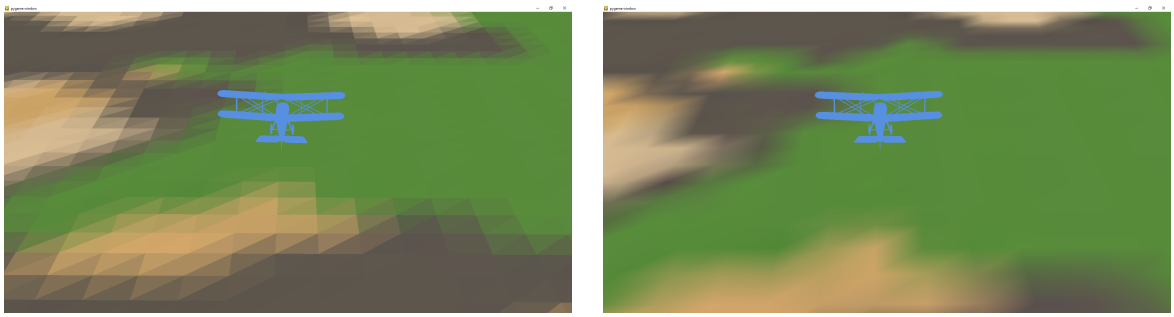


(a) (b)

Figura 8. *grid* do mapa com as primitivas

A última etapa realiza o cálculo das normais e as cores para cada vértice. As normais servem para calcular a influência dos raios de luz nos vértices e consequentemente influenciam na cor que esse vértice vai assumir [Blinn 1977]. Para fazer o cálculo, é necessário fazer uma duplicação dos vértices para cada primitiva, que é um dos métodos utilizados para gerar mapas *low-poly*.

Como varias primitivas compartilham o mesmo vértice no processo de renderização, a normal e cor desse vértice acaba sendo afetado por essas primitivas, o que trás um efeito indesejado de união de texturas, como mostrado na Figura 9, onde a Figura 9 item (a) mostra a textura gerada utilizando o método de duplicação de vértices e a Figura 9 item (b) mostra a textura gerada sem essa duplicação.



(a) (b)

Figura 9. Comparação do método de geração de texturas

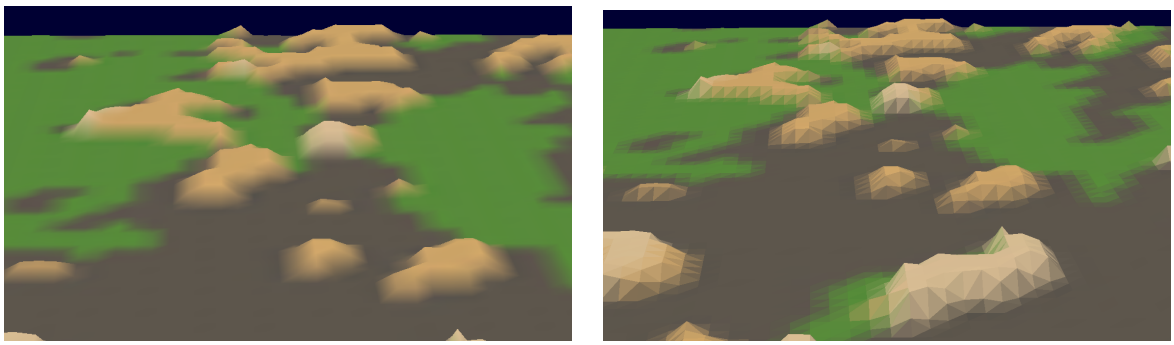
O Algoritmo 3 mostra o processo para duplicação dos vértices e a geração do *vertex buffer* que concentra as informações de coordenadas, cor e normal de cada vértice do terreno. Ele cria para cada primitiva, que no nosso caso é um triângulo, uma cópia dos seus vértices, e a cor de cada um desses vértices é atribuída como a média das cores dos três vértices originais que formam esse triângulo. Já a normal do triângulo é um vetor que faz um ângulo de 90 graus com a superfície desse triângulo e é compartilhada por todos os vértices deste. Ela é calculada como o produto vetorial entre as subtrações das coordenadas desses vértices. Esse cálculo é mostrado na Equação 3, em que Nx, Ny, Nz, são as três coordenadas da normal, já A e B, são respectivamente a subtração vetorial de $p2 - p1$ e $p3 - p1$, sendo $p1, p2, p3$ os pontos que formam o triângulo.

$$(Nx, Ny, Nz) = ((Ay * Bz) - (Az * By), (Az * Bx) - (Ax * Bz), (Ax * By) - (Ay * Bx)) \quad (3)$$

Algorithm 3 Algoritmo para duplicação dos vértices das primitivas

ind_buffer é a lista de índices definidas no algoritmo 2
new_ind_buffer
$colors$ cores dos vértices definidas no algoritmo 1
$vertex\_buffer$ são os dados de coordenadas, cores e normais do *vertex buffer*
$vertices$ é a lista com as coordenadas de todos os vértices do terreno definido no algoritmo 1
for i in $0 : size(ind\_buffe) : 3$ **do**
  $i1, i2, i3 \leftarrow ind\_buffer[i], ind\_buffer[i+1], ind\_buffer[i+2]$
  $color \leftarrow (colors[i1] + colors[i2] + colors[i3])/3$
  $normal \leftarrow cross\_product(vertices[i2] - vertices[i1], vertices[i3] - vertices[i1])$
  $new\_ind\_buffer \leftarrow new\_ind\_buffer + vertices[i1] + color + normal + vertices[i2] + color + normal + vertices[i3] + color + normal$
end for

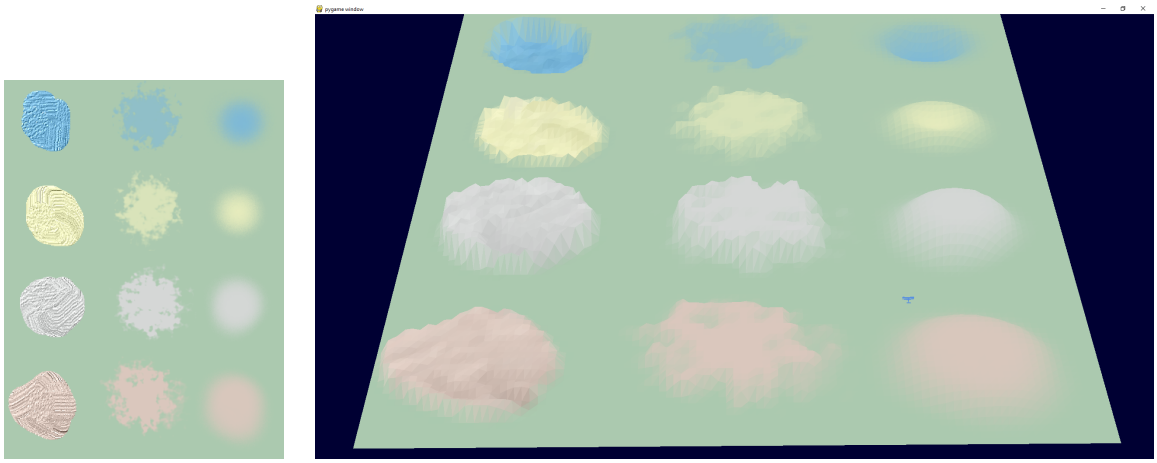
A Figura 10 mostra uma parte do terreno 3D gerado a partir da imagem na Figura 3 e a paleta de cores da tabela na Figura 4. O item (a) mostra o terreno gerado sem a utilização da duplicação dos vértices, já o item (b) mostra o terreno gerado utilizando a duplicação mostrada no Algoritmo 3.



(a) (b)

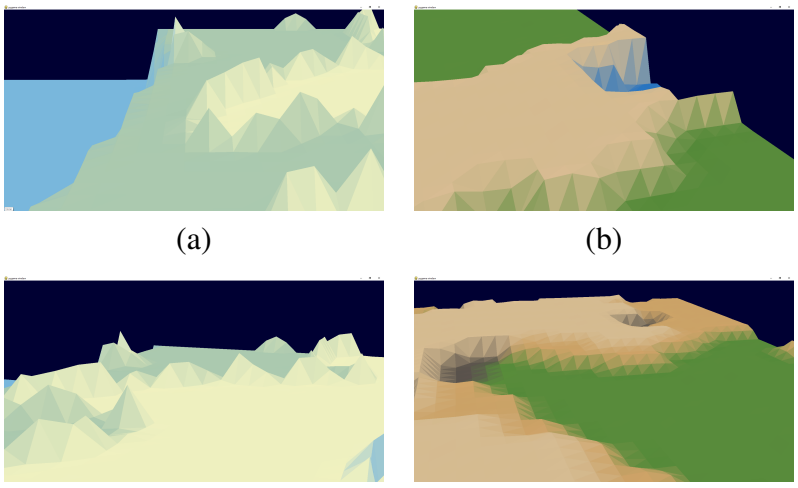
Figura 10. Comparação do método de geração de texturas

Como a técnica proposta utiliza somente a cor dos *pixels* da imagem para gerar o terreno, vários pincéis diferentes na ferramenta de desenho podem ser utilizados pelo usuário para gerar efeitos de terrenos dos mais variados tipos. A Figura 11 mostra o resultado da geração dos terrenos utilizando três pincéis da ferramenta de desenho Paint 3D do sistema operacional windows. Já a Figura 12 mostra o resultado final de alguns terrenos gerados com a nossa técnica.



(a) (b)

Figura 11. Comparação do método de geração de texturas



(a) (b)

(c) (d)

Figura 12. Mapas gerados pelo nosso método

4. Conclusão

Nesse trabalho, nós propomos uma técnica para geração de terrenos 3D low-poly a partir de imagens 2D que podem ser geradas ou desenhadas de forma interativa por um usuário. Essa técnica baseia-se em criar um *grid* de coordenadas 3D do terreno com o mesmo tamanho da imagem escolhida, e a partir das cores dos *pixels* dessa imagem, determinar as alturas ou os valores y das coordenadas de acordo com as cores (r, g, b) dos *pixels* da imagem correspondentes a essas coordenadas.

Aplicações para esse trabalho podem incluir a utilização da abordagem proposta em um editor de terrenos para jogos, com ferramentas de desenho que permitam ao desenvolvedor/designer interagir com a imagem do terreno em tempo real ao mesmo tempo que o mapa 3D do terreno é atualizado de acordo com as alterações na imagem. Outra aplicação possível é a utilização do método proposto para criar terrenos *low-poly* a

partir de imagens geradas por algum outro algoritmo, como o *perlin noise* [Perlin 1985], criando um *pipeline* que facilite o processo de geração de mapas 3D a partir de outros algoritmos.

Existem várias maneiras de avançar com essa pesquisa. Trabalhos futuros poderão incluir a utilização de algoritmos para a detecção automática da paleta de cores, como é o caso do *k-means* [Alsabti et al. 1997], que pode ser utilizado para encontrar essas cores após o usuário determinar o seu tamanho. Outra abordagem que pode ser utilizada para o cálculo da altura (coordenada y) de um vértice no *grid* é a utilização dos vértices ao redor deste, dessa forma o terreno gerado pode sofrer menos mudanças bruscas no relevo, uma vez que estas mudanças fazem diminuir a sensação de realidade do mapa.

Referências

Alsabti, K., Ranka, S., and Singh, V. (1997). An efficient k-means clustering algorithm.

Amato, A. (2017). Procedural content generation in the game industry. In *Game Dynamics*, pages 15–25. Springer.

Angel, E., Shreiner, D., et al. (2012). *Interactive computer graphics: a top-down approach with shader-based OpenGL*. Boston: Addison-Wesley,.

Blinn, J. F. (1977). Models of light reflection for computer synthesized pictures. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198.

Brito, A. (2007). *Blender 3D*. Novatec.

De Carli, D. M., Bevilacqua, F., Pozzer, C. T., and Cordeiro dOrnellas, M. (2011). A survey of procedural content generation techniques suitable to game development. In *2011 Brazilian Symposium on Games and Digital Entertainment*, pages 26–35. IEEE.

Fletcher, M. and Liebscher, R. (2005). Pyopengl–the python opengl binding. *URL: http://pyopengl. sourceforge. net*.

Hung, Y.-P., Chen, C.-S., Hung, K.-C., Chen, Y.-S., and Fuh, C.-S. (1998). Multipass hierarchical stereo matching for generation of digital terrain models from aerial images. *Machine Vision and Applications*, 10(5-6):280–291.

Lutz, M. (2001). *Programming python*. "O'Reilly Media, Inc.".

Marchand, A. and Hennig-Thurau, T. (2013). Value creation in the video game industry: Industry economics, consumer benefits, and research opportunities. *Journal of Interactive Marketing*, 27(3):141–157.

Parberry, I. (2014). Designer worlds: Procedural generation of infinite terrain from real-world elevation data. *Journal of Computer Graphics Techniques*, 3(1).

Perlin, K. (1985). An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296.

Radford, A., Metz, L., and Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

Shreiner, D., Sellers, G., Kessenich, J., and Licea-Kane, B. (2013). *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison-Wesley.