Compressão geométrica de Point Cloud baseada em Dyadic Decomposition

Fundamentos de Compressão de Sinais

Projeto Final

Eduardo Lemos Rocha
Departamento de Engenharia Elétrica
Universidade de Brasília
Brasília, Brasil
170009157

Otho Teixeira Komatsu
Departamento de Engenharia Elétrica
Universidade de Brasília
Brasilia, Brasil
170020142

I. Introdução

Em vídeos de *free-viewpoint* e outras aplicações na computação gráfica, se tornou mais frequente o uso de *point clouds* para representar os objetos sólidos das cenas pela sua facilidade de captura e eficiência computacional de processamento e captura em tempo real em relação aos *meshes* [1], [2].

Essas *point clouds* se tratam de um conjunto de pequenas unidades de representação 3D, que quando são *voxelizadas* passam a ser um conjunto de *voxels*, dotadas de atributos geometricos e atributos de aparência. Uma vez que as *point clouds* são constituídas de centenas de milhares de pontos, se torna essencial o uso de codificadores para sua transmissão e armazenamento [2].

Ao levar isso em consideração, pode-se verificar uma crescente bibliografia explorando algoritmos eficientes de codificação de *point clouds*. Uma técnica recente e que apresenta resultados interessantes, a nível de estado da arte, é a técnica de *dyadic decomposition*, voltada a compressão geométrica de *point clouds* [2], a qual será foco de estudo do projeto.

Essa técnica se baseia em fatiar recursivamente a *point cloud* em 2 metades, gerando uma arvore de recursão em que para cada *point cloud* gerada pela fatia. É construído sua silhueta em um determinado eixo (composição de todos os pontos de cada plano nas coordenadas ao longo desse eixo em uma único plano), e sua silhueta é codificada com base em máscaras provenientes das silhuetas (geradas por essa árvore) previamente codificadas, removendo redundâncias de preenchimento da *point cloud*.

Para o fim da aplicação desse algoritmo, somente os atributos atrelados à geometria são de interesse para a codificação, que são suas localizações nos eixos x, y e z.

II. Implementação

O projeto do compressor geométrico foi desenvolvido em Haskell ghc-8.10.4, constituindo-se do codificador e decodificador. Do codificador espera-se como entrada um arquivo .ply e saída um arquivo .edx que representa os dados da *point cloud* comprimida. Já o decodificador espera-se o arquivo .edx como entrada, produzindo a reconstrução da *point cloud* como saída e expressa em um arquivo .ply(constituindo-se somente dos atributos geométricos).

À nível de codificação e decodificação, o projeto desenvolvido utiliza dos mesmos princípios propostos em [2]. Porém, ao invés de utilizar diretamente uma árvore binária constituída das silhuetas provenientes das *point clouds* repartidas, essa árvore é quebrada e reestruturada em pequenas subárvores de 3 nós, constituídas então pela: silhueta pai, silhueta de seu filho à esquerda e da sihueta do seu filho à direita. Essa estrutura foi denominada de *Triforce*. A imagem abaixo é um exemplo da árvore implementada utilizando essa ideia:

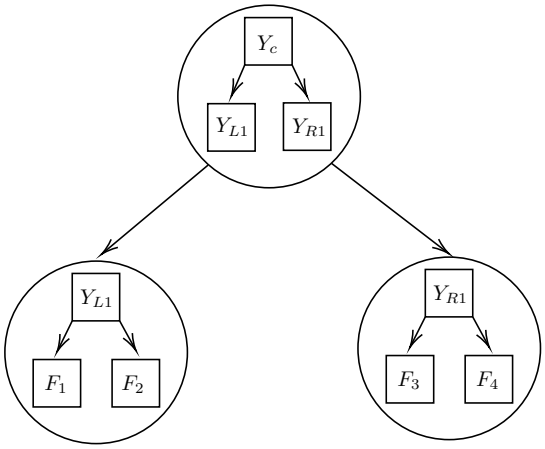

Fig. 1: Árvore de *TriForce*

A partir dessa estrutura, tanto a codificação quanto a decodificação foram realizadas.

A. Codificador

O programa do codificador inicia lendo um arquivo `.ply` através de um *parser* desenvolvido. Com as informações do arquivo coletadas, extrai-se as coordenadas dos *voxels* que ocupam a *point cloud*. Constrói-se a estrutura de dados da *point cloud* (PC) que é definida como um conjunto de *voxels*, em que cada *voxel* representa um conjunto de 3 coordenadas (x, y, z). Além disso, o tamanho da *point cloud*, a maior dimensão em algum eixo onde *voxel* se encontra, também é armazenado nessa estrutura.

A partir dessa PC, cuja dimensão é $\alpha \times \alpha \times \alpha$, a estrutura de árvore de *Triforces* é construída. Dentro do nó raiz, que chamaremos de μ da árvore de *triforce*, há uma pequena estrutura de árvore, formada pela silhueta pai, cuja silhueta cobre o intervalo $[0, \alpha]$; filho à esquerda, $[0, \frac{\alpha}{2})$; e à direita, $[\frac{\alpha}{2}, \alpha])$. No nó do filho à esquerda da raiz mencionada, μ, dessa árvore de *Triforce*; teríamos a mesma estrutura de subárvores, porém sendo o pai representando a silhueta no intervalo de $[0, \frac{\alpha}{2})$, o filho à esquerda $[0, \frac{\alpha}{4})$, e o à direita $[\frac{\alpha}{4}, \frac{\alpha}{2})$. E, finalmente, no nó a direita de μ, temos a estrutura de árvore formada pelas silhuetas que contemplam os intervalos, seguindo a mesma ordem anterior dos nós: $[\frac{\alpha}{2}, \alpha]$, $[\frac{\alpha}{4}, \frac{\alpha}{2})$ e $[\frac{\alpha}{2}, \alpha]$. A mesma ideia se repete no restante da árvore de maneira recursiva, até que os índices da silhueta sejam de tamanho 1, isto é, a silhueta compreende o intervalo $[\frac{\alpha}{n}, \frac{\alpha}{n})$.

A abordagem de codificação descrita no artigo original utiliza uma árvore binária [2] [3] em que são realizadas consultas tanto do nó raiz quanto do filho à esquerda para o mascaramento das silhuetas. Nesse sentido, o principal intuito da estrutura do *Triforce* e de sua árvore na codificação das silhuetas é que ela carregue consigo as informações necessárias ao longo do processo, possibilitando que seja padronizado a ordem de travessia pela árvore. Na nova abordagem escolhida, é sempre priorizado a sub-árvore a esquerda após a codificação da raiz e por último é codificado o nó a direita. Dessa forma, a árvore é codificada com uma travessia *pre-order* [4]. Essa ordem, quando aliada aos recursos da linguagem *Haskell*, como o *fmap* e o *fold*, torna a computação mais eficiente e simples.

O passo seguinte é transformar cada árvore interna de *triforce* em um *bitstream*. Utiliza-se o mesmo procedimento de mascaramento descrito nas referências [2] [3]. A silhueta raiz é usada de máscara para codificar a silhueta da esquerda. A silhueta da direita é feita usando a silhueta da esquerda como máscara. Entretanto, como o nó raiz interno ao *triforce* foi codificado no nó anterior, apenas os nós filhos são codificados. Essa regra apenas não é válida para a raiz da árvore completa de *triforce* que, nesse caso, codifica-se em separado o nó raiz.

Finalmente, concatena-se os binários formados em cada nó da antiga árvore de *triforce* (agora é uma árvore binária de *bitstream*) com a ordem de travessia *preorder*, gerando o *payload* do arquivo que será gerado. Forma-se o arquivo comprimido com a extensão `.edx`. O *header* é composto pela informação do `padding` (zeros adicionados para completar o último *byte*), o eixo em que o *point cloud* foi fatiado e as silhuetas foram formadas, e a dimensão da PC. Seguido do header, vem o *bistream* construído anteriormente.

B. Decodificador

Com o arquivo `.edx` construído, o decodificador é capaz de realizar a descompressão do arquivo, reconstruir a *point cloud*, e a partir disso, reconstruir o arquivo `.ply` com as informações dos atributos geométricos do arquivo original.

Esse processo se inicia com a leitura e *parsing* do arquivo `.edx`, obtendo as dimensões do PC, o eixo na qual as suas fatias de silhuetas foram obtidas e o *bitstream* que representa as informações relativas às silhuetas. Com as informações necessárias para a decodificação, uma árvore de *triforce* é inicializada, cujas *triforces* são compostas por nós que denotam os intervalos (*ranges*) das silhuetas que reconstruirão a *point cloud*. A partir dessa árvore de *triforces* ranges, é obtido a árvore de *triforce* silhuetas a partir da leitura do *bitstream*.

Nesta decodificação, as silhuetas raízes das *triforces* já são previamente obtidas, ou pela leitura da primeira silhueta da *bitstream* (no caso do nó raiz da árvore de *triforce*) ou a partir da decodificação dos nós anteriores. Com essa silhueta pai, Y_C, é construído a silhueta filha à esquerda e à direita seguindo a mesmo mecanismo expresso em [2]. Essas silhuetas, por sua vez, serão as silhuetas raízes das *triforces* seguintes. O processo é recursivamente repetido até obtermos as silhuetas que cobrem um intervalo de tamanho 1, equivalendo-se a uma fatia de espessura 1 da *point cloud*.

A partir dessas fatias, uma vez que elas representam os nós folhas do *triforce* que, ainda, residem nos nós folha da árvore de *triforces*, elas constituem a própria PC, dada a mesma ordem de travessia de árvores já mencionada, *pre-order*, isto é, percorrendo-se os nós na ordem raiz, esquerda e direita. Cada uma dessas fatias representa, então, suas respectivas coordenadas ao longo do eixo de decomposição.

Logo, com os *pixels* dessas fatias, dada as suas respectivas coordenadas ao longo da *point cloud*, formam-se os *voxels* que compunham a PC original. Ao restaurá-la, o programa restaura o arquivo `.ply` original, porém contendo somente os atributos geométricos.

III. Instruções

Para executar o programa proposto, deve-se ter instalado o ambiente *Haskell*, assim como a plataforma *Stack*. Os comandos devem ser executados no diretório raiz do projeto. A compilação do programa é feita utilizando-se do comando abaixo:

```
stack build
```

A execução usufrui de parâmetros passados na linha de comando para prosseguir com a codificação ou decodificação. O seguinte comando informa que será realizado a compressão de um arquivo `.ply`, fatiando a PC ao longo do eixo X:

```
stack exec -- storm-point-cloud-exe -e arquivo.ply X
```

De maneira análoga, o comando que indica decodificação utiliza o nome do arquivo `.edx` como parâmetro:

```
stack exec -- storm-point-cloud-exe -d arquivo.edx
```

TABLE I: Resultado do BPOV das compressões

Files	Tamanho 32			Tamanho 64		
	Original	**Dyadic**	**+PPMD**	**Original**	**Dyadic**	**+PPMD**
Sparse point cloud	105.53	48.61	72/75.07	92.46	88.15	46.30/48.30
Dense point cloud	72.54	1.96	2.09/2.07	76.26	1.99	1.96/1.92

IV. Resultados

A tabela acima contém os resultados obtidos a partir de *point clouds* geradas randomicamente. Para cada tamanho de PC, uma versão distribuída (*Sparse*), com poucos *voxels*, e outra densa (*Dense*), com um grande número de pontos, foram construídas. Para as distribuídas, 80% do tamanho foi utilizado como número de pontos existentes. Em contrapartida, as versões densas foram compostas aproximadamente por 50% do volume da PC em pontos. A unidade escolhida para comparação em todos os casos foi a *BPVO*, ou *bits per voxel*.

Além disso, foram feitas compressões com o compressor `7Zip` (método PPMD) após o processo de codificação da PC ter sido concluído, isto é, é comprimido o arquivo `.edx` em um `.zip`. Foi sugerido, para obter um melhor resultado nessa segunda etapa de compressão, que os *bits* do `.edx` fossem expandidos para um *byte*, visto que o método de compressão escolhido no `7Zip` se beneficia dessa adaptação. Os resultados mostram os valores com e sem a expansão. Os parâmetros padrão foi usados na ferramenta.

V. Conclusão

Com base nos resultados obtidos, nota-se que o compressor geométrico desenvolvido apresenta melhores resultados com *point clouds* mais preenchidas. A remoção de redundâncias se torna mais efetiva com mais pontos sendo inferidos pelo decodificador. Com poucos *voxels* na PC, vários *bits* são desperdiçados ao enviar a primeira silhueta completa, causando pouco ganho pela compressão. Pelo mesmo motivo, estima-se que em tamanhos maiores expansões ocorram no arquivo final.

Ademais, a adesão de uma segunda codificação sem perdas (`7Zip` com método PPMD) após executar o compressor diático resultou em piores resultados, quando comparado ao uso isolado do trabalho descrito. Entretanto, aplicar a mesma metodologia em *point clouds* densas mostraram indícios de uma melhoria progressiva no resultado final. Entende-se que com o aumento dos tamanhos, os ganhos seriam cada vez maiores.

References

[1] R. L. de Queiroz and P. A. Chou, "Compression of 3D Point Clouds Using a Region-Adaptive Hierarchical Transform," in IEEE Transactions on Image Processing, vol. 25, no. 8, pp. 3947-3956, Aug. 2016, doi: 10.1109/TIP.2016.2575005

[2] E. Peixoto, "Intra-Frame Compression of Point Cloud Geometry Using Dyadic Decomposition," in IEEE Signal Processing Letters, vol. 27, pp. 246-250, 2020, doi: 10.1109/LSP.2020.2965322.

[3] D. R. Freitas, E. Peixoto, R. L. de Queiroz and E. Medeiros, "Lossy Point Cloud Geometry Compression Via Dyadic Decomposition," 2020 IEEE International Conference on Image Processing (ICIP), 2020, pp. 2731-2735, doi: 10.1109/ICIP40778.2020.9190910.

[4] https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/