6
Resultados

Este capítulo descreve os resultados das otimizações descritas na seção 3.3, apresentando apenas testes feitos com $d = 3$, isto é, no espaço tridimensional. Comparações com uma implementação clássica de uma *octree* foram feitas para diferentes formas de busca. Modelos de teste, modelos de busca e modelos de comparação são descritos abaixo. A 2^d–*tree* aberta usada nos testes para validar as otimizações foi descrita de forma geral nas seções 4.1 e 2.5.3. Os testes foram rodados numa máquina com uma configuração contendo um Pentium IV de $3GHz$ e 2 Gb de RAM.

6.1
Modelo de Teste

Foram testadas as otimizações propostas na seção 3.3 em uma *hashed octree* adaptada num conjunto de pontos da seguinte forma: os nós são subdivididos até conterem apenas um ponto do conjunto ou se a *hashed octree* atingir um nível máximo, que foi estipulado como 21, devido ao uso de armazenamento de chaves de tamanho 64-bits no espaço tridimensional. A função de *hash* usada foi a mesma para todos os modelos e para todos os tipos de busca retornando os 21 bits menos significantes da chave. Foi constatado que tal função de *hash* tende a distribuir de maneira eficiente as chaves na *hash table*, mesmo para dados propícios a desbalancear uma *octree* clássica.

6.2
Modelo de Comparação

Os resultados foram comparados com os procedimentos de busca clássicos (seção 2.3) em um armazenamento clássico na forma de árvore, mas em duas implementações diferentes: uma implementação clássica (oito ponteiros) e outra implementação filho/irmão (seção 2.5.2). Os resultados foram também comparados com um armazenamento na forma de uma *hash table* (seção 2.5.3), mas sem as otimizações propostas na seção 3.3. Foi verificado que buscas num armazenamento na forma de uma *hash table* sem otimização são mais lentas,

	f_{pontos}	f_{folhas}	f_{grade}
Busca por Pontos	**1.30**	1.33	2.45
Busca por Folhas	0.95	**0.93**	1.52
Busca numa Grade	1.70	1.66	**0.89**

Tabela 6.1: Média em todos os conjuntos de pontos do tempo de execução (em microssegundos) da busca direta, simulando diferentes modelos de freqüência (linhas). Otimizando de acordo com a freqüência correta do modelo (termos diagonais) melhora-se o método de busca 2% (pontos) e 92% (grade). O número de buscas em cada caso está descrito na tabela 6.3.

já que o acesso ao filho de um nó exige um acesso à *hash table*, que é mais lento do que o acesso direto ao filho de um nó via ponteiros.

6.3 Modelo de Buscas Diretas

Três tipos de busca diretas foram usados para variar a freqüência de acesso *a priori* f_n com que uma folha de nível n é buscada, como descrito na seção 3.3, a fim de testar o modelo de custo em diferentes cenários. Foram usados três modelos de freqüência: buscando por cada ponto do conjunto, onde a freqüência de cada folha é definida como sendo a quantidade de pontos dentro dela: ($f_{pontos}(n) = \#\{pontos \in n\}$); buscando por cada folha da *octree*, onde a freqüência é definida como sendo 1 para cada folha: ($f_{folhas}(n) = 1$); e por cada célula da grade regular 127^3, criada na região definida pela *octree* e buscada cada célula da grade uma vez: ($f_{grade}(n) = 2^{3(7-l(n))}$). Essas otimizações estão experimentalmente validadas na tabela 6.1: as buscas foram significantemente rápidas onde foi escolhida a freqüência correta do modelo para a otimização.

6.4 Modelo de Objetos Geométricos

Os objetos geométricos usados para os testes são classificados em quatro categorias: densidade altamente variável, a qual tende a linearizar o tempo das buscas numa *octree* clássica; dados volumétricos conhecidos que preenchem todo o domínio; dados randômicos, seguindo uma distribuição uniforme dos dados; superfícies de grande extensão. Para dados de densidade altamente variável, as *octrees* de ponteiros têm uma tendência a ficar desbalanceadas, e por isso os tempos de busca são bem piores do que para um modelo do mesmo tamanho melhor distribuído. Devido à função de *hash* escolhida, isso não acontece com a *hashed octree*. Ela não fica desbalanceada, e o tempo de busca para alguns modelos se comportou mais de dez vezes mais rápido do que a *octree* clássica (modelo *Maracanã* na 6.2). Para dados volumétricos, que são

distribuídos no domínio todo, a *hashed octree* se comportou na média quatro vezes mais rápida do que a *octree* clássica. E em dados grandes de superfícies, como, por exemplo, o dado *Happy Budda*, a *hashed octree* foi mais de três vezes mais rápida.

Para modelos muito grandes, como, por exemplo, o modelo *Thai Statue* que possui 5.000.000 pontos, onde foram gerados 22.296.000 nós na *octree*, a *hashed octree* encontrou um pouco de dificuldade, ganhando por pouco em tempo. Isso acontece por causa do grande número de nós gerados, exigindo uma lista de colisão na *hash table* com vários níveis, com o fator de balanço aproximadamente $\frac{1}{10}$, pois o número de entradas da *hash table* é de 2.097.152 para uma função de *hash* retornando os 21 bits menos significantes, fazendo com que a busca deixe de ser tão eficiente devido a uma grande parte dela conter acessos seqüenciais da lista de colisão. Seria necessário aumentar o número de bits de retorno da função de *hash* para aumentar o número de entradas e assim diminuir alguns níveis da lista de colisão, a fim de evitar a influência do acesso seqüencial nas buscas. Mas note que a memória consumida pela *hashed octree* para modelos grandes foi metade da consumida pela *octree* clássica e a mesma quantidade consumida pela *octree* filho-irmão, como por exemplo, o modelo *Thai Statue*: a *octree* clássica gastou 1.070 MB de memória para armazenar os nós da *octree*, enquanto a *octree* filho-irmão e a *hashed octree* gastaram 535 MB. Para esse objeto, o método chegou num extremo da relação de eficiência entre tempo de busca e memória consumida, gastando metade da memória e ainda assim ganhando de 0.3 vezes no tempo de busca, comparado com a *octree* clássica. Agora, se comparado com a *octree* filho-irmão, as duas consomem a mesma quantidade de memória, mas a *hashed octree* é quase seis vezes mais rápida na busca direta por pontos. Com as otimizações feitas na *hashed octree*, o usuário controla a relação de eficiência entre tempo de busca e memória consumida pela função de *hash*. Todos os dados podem ser conferidos na tabela 6.2. E para os níveis e tempos de pré-processamento, conferir a tabela 6.3.

Conjunto de Pontos	**Busca Direta**				**Busca por Vizinhos Adjacentes**				**Busca por Vizinhos num Raio**			
	8_{ptr}	**F-I**$_{ptr}$	**Desopt**	**Nosso**	8_{ptr}	**F-I**$_{ptr}$	**Desopt**	**Nosso**	8_{ptr}	**F-I**$_{ptr}$	**Desopt**	**Nosso**
Sphere	2.17	7.24	11.47	0.58	23.10	23.85	41.55	7.63	28.38	25.39	41.49	0.71
Bunny	2.86	11.28	18.67	0.75	38.34	35.93	59.70	8.70	36.70	35.44	59.26	0.73
Maracanã	6.53	31.21	53.47	0.60	58.98	55.79	95.70	9.12	77.75	76.31	136.91	33.72
David head	3.29	12.98	21.27	0.64	43.57	40.88	68.69	9.43	42.61	40.93	70.30	1.39
Pig	3.71	15.40	25.65	0.71	46.46	43.90	74.64	11.73	53.53	51.59	91.88	10.67
CSG	3.30	13.34	21.98	0.79	40.60	37.75	63.22	9.76	38.79	37.18	62.81	4.57
Volcano	3.28	13.42	22.73	0.71	43.04	40.37	68.74	8.83	40.18	39.43	69.16	2.78
Fighter	3.89	14.82	24.53	1.06	54.70	50.38	86.60	12.88	51.81	49.92	87.79	2.18
Deltao	4.47	18.63	31.64	1.65	49.07	45.56	77.67	11.30	180.30	169.13	332.11	129.28
David	3.58	15.04	26.54	0.87	46.13	42.64	73.64	10.39	43.71	42.17	74.79	11.88
Rnd Sphere	4.50	16.42	27.92	1.59	46.51	42.55	73.15	14.45	42.43	41.42	72.71	9.02
Dragon	3.81	15.88	27.43	1.17	45.53	42.80	74.31	12.79	45.88	44.44	79.03	11.67
Happy	3.80	16.17	28.69	1.21	46.30	43.05	75.37	13.18	48.60	46.39	83.23	15.46
Blade	3.65	15.25	27.98	1.27	44.05	40.89	73.09	13.11	43.84	41.54	75.27	11.45
Chair	3.83	16.38	31.41	2.15	46.56	43.43	80.89	19.34	52.92	50.80	96.60	24.48
Rnd Cube	4.18	14.76	33.54	3.79	51.36	46.69	95.21	27.44	45.34	44.04	84.09	0.96
Thai Statue	3.93	17.38	37.51	3.13	49.99	47.58	94.86	22.43	79.50	77.18	163.46	53.56
média	**3.81**	**15.62**	**27.79**	**1.33**	**45.55**	**42.59**	**75.12**	**13.09**	**56.02**	**53.72**	**98.88**	**19.09**
ganho	× **3.8**	× **15.7**	× **26.9**	× 1	× **3.8**	× **3.6**	× **6.2**	× 1	× **14.5**	× **13.8**	× **24.2**	× 1

Tabela 6.2: Tempo de execução (em microssegundos) de nossas buscas otimizadas, comparada com buscas clássicas de cima para baixo numa *octree* clássica 8-ponteiros (8_{ptr}), numa *octree* filho-irmão (F-I$_{ptr}$) e numa *hashed octree* sem otimização (*Desopt*). Nossa otimização na busca direta retornou 15 vezes mais rápida do que a filho-irmão, e entre 10% (volume randômico) e 1090% mais rápida (*Maracanã*) do que a *octree* clássica. Cada tempo representa a média do tempo das buscas por cada ponto do conjunto. O ganho corresponde à razão do tempo de busca por ponteiro sobre o nosso tempo de busca. O tamanho do raio usado foi de 0.1 da normalização dos dados.

Conjunto de Pontos	**# Nós** (x1000)	**Profundidade Nível**	**Memória** $\mathbf{8}_{ptr}$ (MB)	**F-I**$_{ptr}$ (MB)	**Hash** (MB)	**Construção** $\mathbf{8}_{ptr}$ (msec)	**F-I**$_{ptr}$ (msec)	**Hash** (msec)	**Otimização Tempo** (msec)	**Nível**	**Número de Buscas Ponto** (x1000)	**Folha** (x1000)
Sphere	4.4	6	0.2	0.1	45	5.6	5.4	11.1	7.8	4	0.9	3.8
Bunny	144	9	6.9	3.4	50	224	215	187	14.2	7	35	126
Maracanã	190	21	9.1	4.6	51	268	257	230	27.1	21	10	167
David head	395	13	19	9.5	51	726	682	548	32.4	9	100	346
Pig	401	11	19	10	51	772	725	583	38.2	10	100	351
CSG	529	19	25	13	52	819	768	642	50.7	8	82	463
Volcano	551	21	26	13	52	1 043	967	795	50.2	9	133	482
Fighter	723	14	35	17	54	1 584	1 509	1 211	58.6	9	257	632
Deltao	1 210	20	58	29	60	2 466	2 119	1 695	116	12	208	1 058
David	1 534	21	74	37	62	3 124	2 819	2 379	135	10	350	1 342
Rnd Sphere	2 883	18	138	69	82	5 756	5 100	4 053	268	9	500	2 523
Dragon	3 060	21	147	73	85	5 634	5 280	4 286	314	10	438	2 678
Happy	3 617	21	174	87	96	7 227	6 627	5 361	377	10	544	3 165
Blade	5 412	16	260	130	134	11 641	10 079	8 384	518	10	883	4 735
Chair	9 255	18	444	222	223	34 471	24 833	20 076	930	11	1 669	8 098
Rnd Cube	15 675	12	752	376	376	45 867	39 380	31 012	1 202	8	4 000	13 715
Thai Statue	22 296	20	1070	535	535	155 803	106 509	81 446	2 229	12	5 000	19 509
	média ponderada		**632**	**316**	**321**	**68 436**	**49 213**	**38 084**	**354**			

Tabela 6.3: Tempos de construção e pré-processamento (em microssegundos) em diferentes conjuntos de pontos: densidade altamente variável (Maracanã, David head, Pig, Fighter, Deltao), volumétrico (Volcano, Fighter, Deltao, Cube), Randômico (Rnd Sphere, Rnd Cube). A construção da *hashed octree* é ligeiramente mais rápida do que a da *octree* com ponteiros. O pré-processamento consiste do cálculo do nível ótimo e representa menos de um por cento do tempo de construção. As médias são ponderadas pelo tamanho da *octree*. O número de buscas está relacionado com cada ponto do conjunto de acordo com o modelo de freqüência simulado (busca numa grade tem um número constante de buscas 2097).