4
Implementação em CPU

Este capítulo tem por finalidade avaliar os algoritmos e técnicas de aceleração levantadas anteriormente, a fim de obter a melhor combinação entre eles avaliando o desempenho conseguido em cada um dos caminhos de câmera pelos modelos. Ao final do capítulo pretende-se obter o estado da arte do algoritmo de *frustum culling* juntamente com as suas técnicas de aceleração.

4.1
Renderização

A renderização de modelos massivos, mesmo com as placas gráficas mais modernas, ainda é um desafio e necessita de algoritmos especiais para que possam ser viáveis. Segundo [16] as GPUs mais recentes são capazes de processar de 10 a 200 milhões de polígonos por segundo, assumindo que a geometria já esteja na memória da GPU. Mesmo as GPUs mais modernas não possuem espaço necessário para alocar modelos massivos. Mesmo se coubessem, a renderização de um *frame* do Boeing 777, por exemplo, levaria 1,75 segundos, que é bem longe da taxa iterativa de 0,33 segundos (30 *frames* por segundo).

Como o foco deste trabalho é estudar técnicas de *frustum culling*, e não foram implementadas técnicas avançadas para viabilizar a renderização dos modelos massivos em taxas iterativas, na maioria dos testes a renderização foi desligada, a fim de facilitar a localização dos gargalos. Assim, durante o caminho de câmera apenas o algoritmo de *frustum culling* é realizado. Os testes de *frustum culling* com as GPU *primitives* foram os únicos que tiveram a renderização ligada.

4.2
Radar X Planos

A Seção 3.2.2 levantou o descarte de volumes envolventes, diferente da abordagem tradicional por planos, conhecido como radar. O trabalho [66] faz a comparação entre as duas abordagens e mostra o ganho de performance do radar. Porém não são descritos detalhes de implementação, como a utilização de hierarquia e os cálculos de descarte das AABBs.

A primeira análise entre essas duas técnicas é referente à atualização dos dados da câmera. A abordagem clássica precisa extrair a equação dos planos do *frustum* de visão, como foi visto anteriormente, o que demanda uma série de cálculos. Em contrapartida, a abordagem do radar só precisa calcular os limites da câmera uma vez no início da aplicação, ou quando seus parâmetros são modificados, e depois é suficiente atualizar seus vetores (*up*, *right* e *forward*) quando a câmera muda sua posição ou orientação. Esta atualização quando comparada com a extração de planos mostra-se bastante eficiente. A segunda análise refere-se aos cálculos de descarte. A abordagem do radar realiza menos operações que a abordagem clássica como pode ser visto na Tabela 4.1, onde são expostos os números de operações necessárias para atualização e descarte de volumes envolventes no melhor e pior caso. Estes números foram conseguidos contabilizando cada uma das instruções do código gerado, atribuindo pesos de acordo com a instrução executada [71].

Método	# Atualizações	Melhor caso	Pior caso
Planos	316 operações	7 operações	336 operações
Radar	112 operações	8 operações	192 operações
Planos+Otimização	316 operações	10 operações	120 operações

Tabela 4.1: Radar X Planos.

Analisando os algoritmos de planos e radar, a escolha mais correta seria adotar o radar como representante do algoritmo de *frustum culling*, porém quando é incorporado ao algoritmo de planos a otimização onde apenas 2 vértices são testados contra os planos [30, 27] (última linha da Tabela 4.1), seu desempenho aumenta tendo um melhor caso médio. A atualização dos atributos dos algoritmos, onde o radar levou ampla vantagem, quando comparado com os cálculos de descarte dos volumes envolventes, consomem uma parte ínfima do processamento e não precisa ser levado em conta. A diferença de performance dos três algoritmos citados fica clara nos testes realizados com os caminhos de câmera nas plataformas como pode ser visto na Figura 4.1.

O algoritmo de planos juntamente com a otimização obteve melhor desempenho em todo o caminho de câmera das plataformas de petróleo. Tendo melhor desempenho, o caminho de câmera é feito mais rápido e com isso em alguns casos o gráfico termina antes que os outros como por exemplo na P-43. O comportamento de simetria nos gráficos dos algoritmo de radar e planos com otimização pode ser explicado pelo fato de que o ponto forte de um algoritmo é o ponto fraco do outro. Enquanto o algoritmo de planos com otimização consegue detectar que um volume envolvente está fora do *frustum* de maneira rápida, o algoritmo de radar precisa realizar mais cálculos para chegar a este

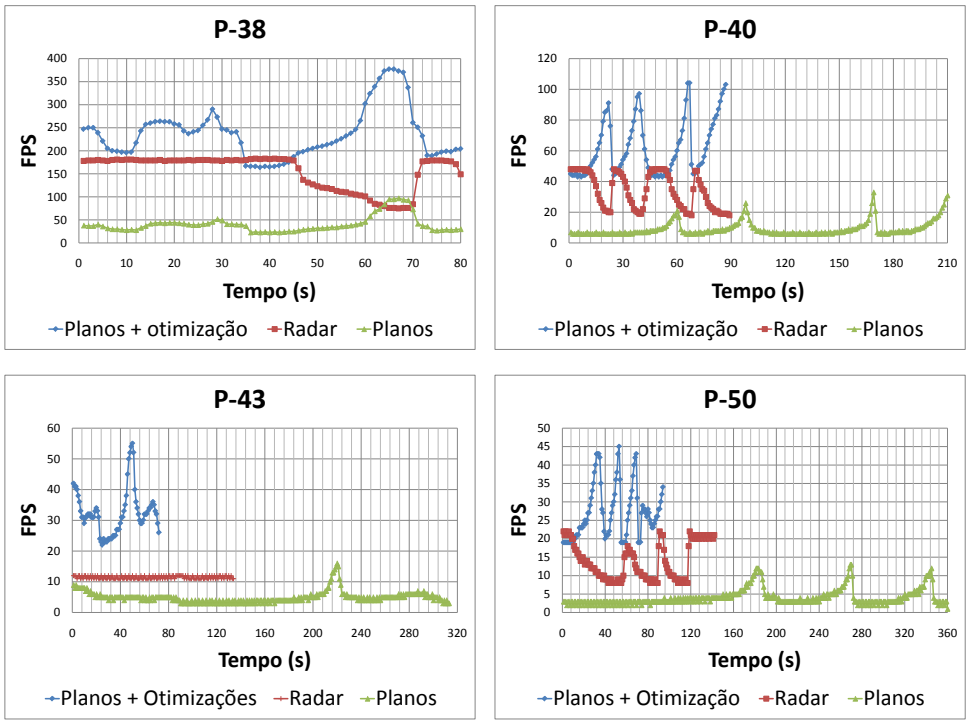


Figura 4.1: Caminhos de câmera sem hierarquia nas plataformas.

resultado. No caso do Boeing, o radar superou o algoritmo de planos com otimização como pode ser visto na Figura 4.2.

A execução dos três algoritmos, nesses testes, não utilizou hierarquia, e com isso não houve a necessidade de retornar o resultado de interseção dos volumes com o *frustum*. Quando há a necessidade de retornar possíveis interseções, quando a hierarquia é utilizada, o algoritmo de radar implementado perde performance e foi superado no único caminho de câmera que ainda faltava, como pode ser visto na Tabela 4.2.

Método	Frames Processados	FPS Mínimo	FPS Máximo	Média do Caminho
Radar sem interseção	2674	34	37	36.217
Radar com interseção	1553	15	17	16.096
Planos + Otimização	2522	32	39	34.171

Tabela 4.2: Radar sem interseção X Radar com interseção para o Boeing.

Como a utilização de hierarquia é primordial para o desempenho do algoritmo, a abordagem que utiliza planos com a otimização de testar apenas dois vértices da AABB foi escolhida como algoritmo de descarte de volumes envolventes.

Neste momento o gargalo da aplicação encontra-se no descarte dos volumes devido à grande quantidade de cálculos executados. As próximas seções

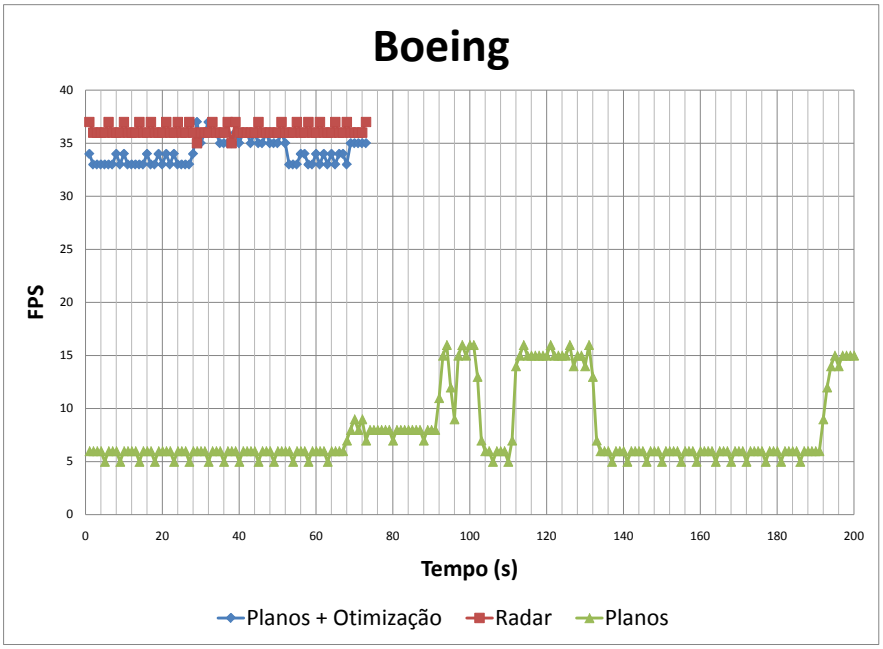


Figura 4.2: Caminhos de câmera sem hierarquia no Boeing.

tentam incorporar as otimizações descritas anteriormente para minimizar os efeitos da inserção do algoritmo de *frustum culling* na aplicação.

4.3 Hierarquia

Na Seção 3.2.1 foram levantados vários tipos de volumes envolventes, onde, à medida que se ajustam melhor com os objetos, demandam maior processamento na execução dos algoritmos. Também foi decidido que a AABB seria o volume envolvente utilizado neste trabalho. Para acelerar os cálculos do *frustum culling* foi introduzida a ideia de hierarquia de volumes envolventes. Ficou para ser decidido qual seria a melhor forma de construção da hierarquia e qual a estrutura de particionamento do espaço seria utilizada. Dada a diversidade e complexidade das cenas, não existe heurística ideal para todos os modelos.

Foram construídas hierarquias do tipo *top-down* com o particionamento do espaço, utilizando a média e a mediana dos centróides das caixas envolventes. Todos os modelos foram submetidos aos diferentes tipos de construção de hierarquia a fim de analisar qual delas proporciona a melhor árvore. Neste caso, a melhor árvore é a que resultar em melhor *fps* nos seus respectivos caminhos de câmera. A Tabela 4.3 ilustra os resultados obtidos com a construção de hierarquia utilizando mediana.

A construção da hierarquia utilizando mediana obteve árvores de al-

Modelos	# Nós	# Folhas	Altura	Tempo de construção(s)
P-38	174533	87267	20	3.29
P-40	559303	279652	22	3.64
P-43	1104911	552456	23	7.70
P-50	1119449	559725	23	8.92
Boeing 777	1383049	691525	21	7.37

Tabela 4.3: Hierarquias utilizando mediana.

tura menor quando comparadas com as hierarquias obtidas utilizando média (Tabela 4.4). A construção utilizando mediana demorou mais tempo em virtude da necessidade de calcular a mediana dos pontos ao longo do maior eixo, o que envolveu um algoritmo de ordenação.

Modelos	# Nós	# Folhas	Altura	Tempo de construção(s)
P-38	225833	112917	24	2.66
P-40	756209	378105	26	2.81
P-43	1388287	694144	27	5.65
P-50	1541347	770674	28	6.84
Boeing 777	1390645	695323	29	4.85

Tabela 4.4: Hierarquias utilizando média.

A construção das hierarquias adotou como único critério de parada, a existência de três objetos de cada tipo. Dessa forma um nó folha terá no máximo três identificadores de geometria de cada tipo. Esta restrição foi imposta a fim de obter árvores com maior altura e consequentemente forçar mais cálculos em determinados *frames*, o que ajuda na determinação do algoritmo de melhor eficiência. As Figuras 4.3 e 4.4 mostram os caminhos de câmera feitos com o algoritmo de planos e a otimização de testes de apenas dois vértices nos dois tipos de hierarquias construídas.

Os caminhos de câmera para as plataformas P-38 e P-50 e o Boeing (Figura 4.3) tiveram melhor performance nas hierarquias construídas utilizando média dos centróides.

A Tabela 4.5 mostra com mais detalhes as performances dos caminhos de câmera ao longo das hierarquias construídas para a P-38 e P-50 e o Boeing.

Os caminhos de câmera nas plataformas P-40 e P43 (Figura 4.4) obtiveram melhor performance utilizando hierarquias construídas a partir da mediana dos centróides dos volumes envolventes. A Tabela 4.6 mostra com mais detalhes as suas performances.

Apesar da diferença entre a performance dos dois tipos de hierarquia ser pequena, ela foi levada em conta e os que obtiveram melhor desempenho foram

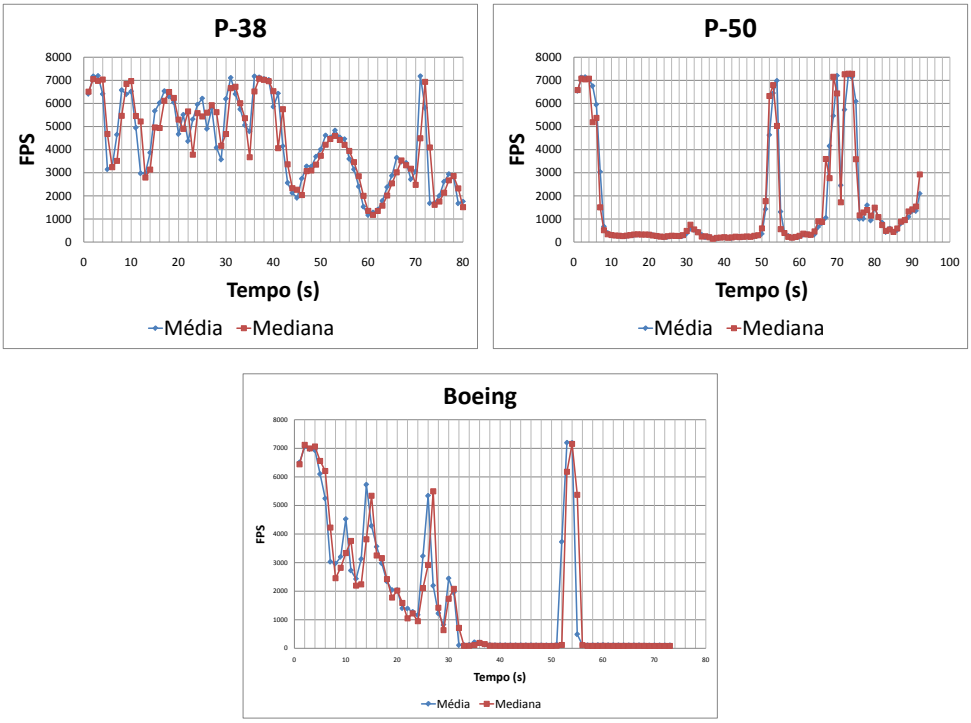


Figura 4.3: Caminhos de câmera com hierarquias onde média foi melhor.

utilizados em conjunto com os demais algoritmos.

Uma forma de otimizar o tempo de processamento do algoritmo de *frustum culling* é interromper o percurso da hierarquia a partir de certa altura da árvore. Esse tipo de otimização não foi feita pois a ideia deste trabalho é buscar o algoritmo que obtenha melhor desempenho, porém retornando o mínimo de geometrias não visíveis.

4.4 Percurso

O percurso da hierarquia sem a utilização de pilha trouxe dois grandes benefícios à aplicação. A primeira foi a redução do uso de memória, de suma importância em modelos massivos. No pior caso, a pilha teria o tamanho da altura da árvore. O uso de memória ainda pode ser maior quando são utilizados vários processadores no percurso, como será visto na Seção 4.7.

A outra vantagem obtida com a utilização do percurso sem pilha é o ganho de performance. A utilização de pilha implica em três possibilidades de alocação de memória. A primeira é alocar estaticamente o máximo de memória que pode ser utilizada no percurso, o que agravaria mais ainda o problema diagnosticado anteriormente, e descobrir o mínimo de memória que será utilizado previamente é uma tarefa difícil. A segunda seria fazer uma alocação dinâmica de memória, o que implica em introduzir um *overhead* no

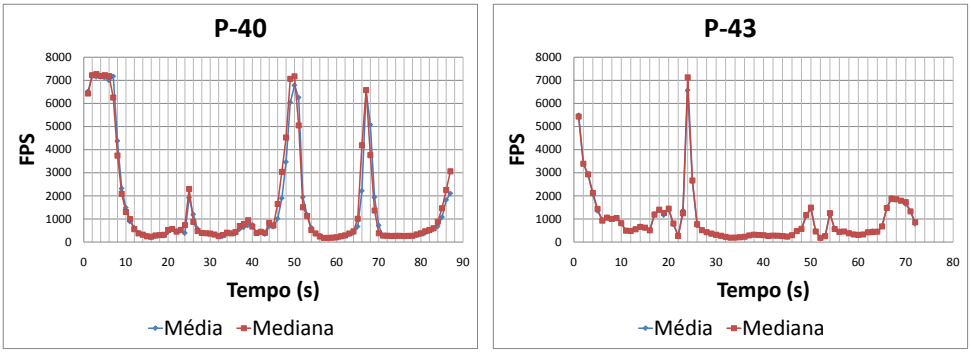


Figura 4.4: Caminho de câmera com hierarquias onde mediana foi melhor.

Método	Frames Processados	FPS Mínimo	FPS Máximo	Média do Caminho
Mediana na P-38	347447	**1168**	7101	4296.6
Média na P-38	**348543**	1152	**7200**	**4345.7**
Mediana na P-50	144655	**142**	**7303**	1570.5
Média na P-50	**146609**	137	7274	**1584.4**
Mediana no Boeing	127491	84	**7235**	1727.7
Média no Boeing	**129179**	**103**	7215	**1766.8**

Tabela 4.5: Detalhes dos caminhos com hierarquias de média e mediana.

Método	Frames Processados	FPS Mínimo	FPS Máximo	Média do Caminho
Média na P-40	138886	169	7232	1580.7
Mediana na P-40	**142160**	**179**	**7284**	**1625.3**
Média na P-43	68456	**182**	6577	943.0
Mediana na P-43	**69493**	178	**7153**	**957.2**

Tabela 4.6: Detalhes dos caminhos com hierarquias de média e mediana.

algoritmo. A terceira utilizaria recursão e não foi implementada. Os ganhos em termos de desempenho na remoção do uso de pilha podem ser observados nas Figuras 4.5 e 4.6.

Em todos os testes a remoção da necessidade de pilha no percurso da hierarquia obteve ganhos que variaram de 1.5% no caso da plataforma P-50 até 11.69% no caso do Boeing. Os testes foram feitos utilizando o algoritmo de planos com a otimização de testes de apenas dois vértices e as hierárquias que obtiveram melhor performance em cada modelo.

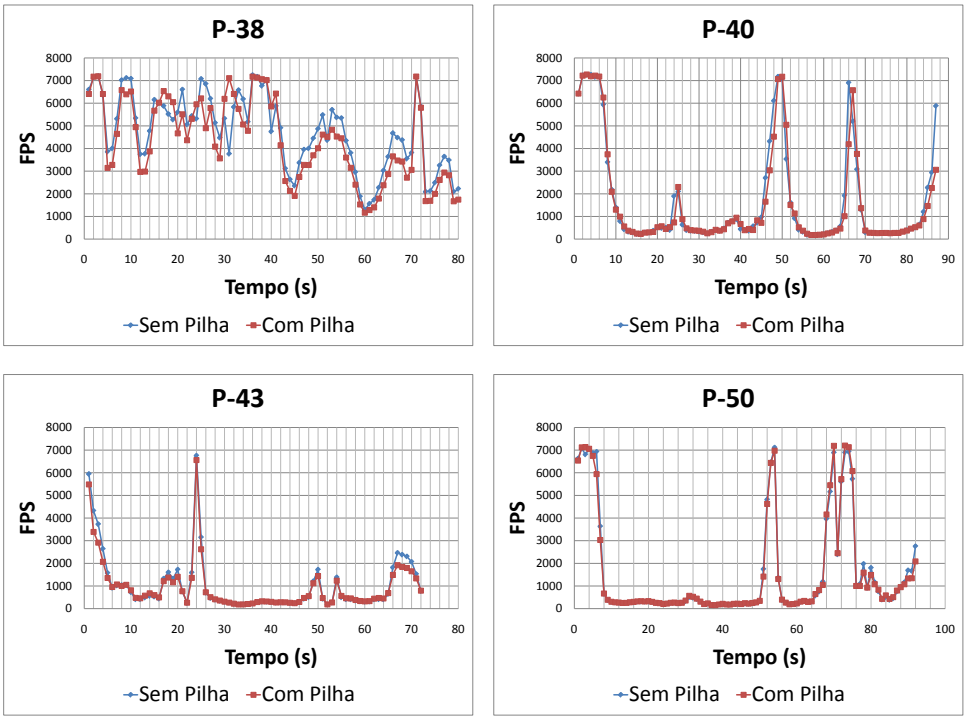


Figura 4.5: Caminho de câmera sem pilha nas plataformas.

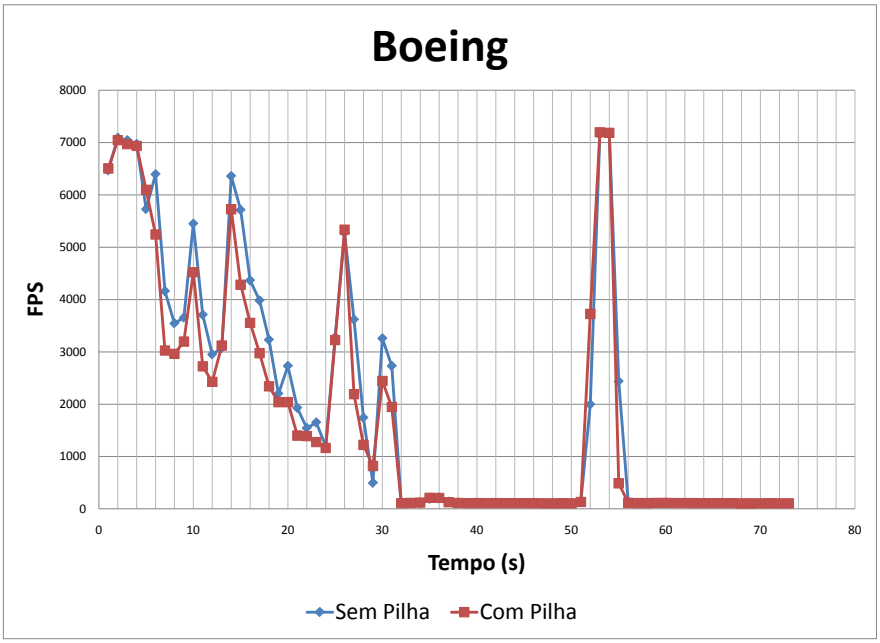


Figura 4.6: Caminho de câmera sem pilha no Boeing.

4.5 Otimizações

A primeira otimização a ser inserida no *pipeline* do *frustum culling* foi a remoção da necessidade de testar todos os oito vértices da AABB contra os planos. Como foi visto na Seção 4.2 esta otimização aumentou o desempenho da aplicação em duas vezes quando comparado ao algoritmo de radar e em quatro vezes em relação a implementação de planos sem otimização.

A conclusão do trabalho de Assarsson e Möller *et al.* [3], citados anteriormente, identificou que a combinação entre as otimizações *plane-coherency test* e *octant test* obtiveram a melhor performance, como pode ser visto na Tabela 4.7.

Path	1	1	1	2	2	2	3	3	3	4	4	4
Model	1	2	3	1	2	3	1	2	3	1	2	3
Only Basic intersection test	2.8	1.9	3.9	2.2	2.0	3.1	3.9	2.5	4.3	3.1	2.2	3.7
Plane-coherency + octant test	**4.0**	**2.4**	**5.1**	**2.8**	**2.6**	**3.9**	4.8	**3.5**	**5.6**	3.3	3.0	5.1
Plane-coherency + TR coherency	3.8	2.0	4.0	2.5	2.2	3.0	5.0	2.8	4.4	**8.3**	3.1	**11.0**
Plane-coherency + octant test + TR coherency	3.7	2.2	4.5	2.6	2.4	3.6	**5.1**	3.0	4.8	8.0	**3.3**	9.0

Tabela 4.7: Resultados obtidos no trabalho de Assarsson.

A Tabela 4.7 mostra os resultados obtidos por Assarsson fazendo diferentes combinações de otimizações em três modelos de tamanhos variados contendo quatro camanhos de câmera diferentes. Segundo Assarsson, a otimização *masking* não se mostrou competitiva com as outras. Seus valores estão indicando o quanto os algoritmos foram mais eficientes quando comparados com o caminho de câmera sem teste de interseção. A inserção desses dois algoritmos no *pipeline* do *frustum culling*, neste trabalho, obteve os resultados expressos na Figura 4.7 para as plataformas e na Figura 4.8 para o Boeing.

A utilização das duas otimizações aumentou a performance da aplicação em todos os casos de testes quando comparada com o percurso da hierarquia no caminho de câmera sem nenhuma otimização.

A última otimização citada na Seção 3.2.4, convertia o *frustum* de visão em uma AABB e realizava testes entre duas AABB para determinar se o objeto estaria totalmente fora do *frustum* de forma mais rápida do que o teste convencional. Essa otimização eleva o tempo de processamento da atualização da câmera, pois é necessário determinar a AABB da câmera, porém diminui o esforço gasto no descarte dos volumes envolventes em algumas situações. A determinação da AABB da câmera quando comparado com os testes de descarte pode ser ignorado em termos de tempo de processamento. Apenas os

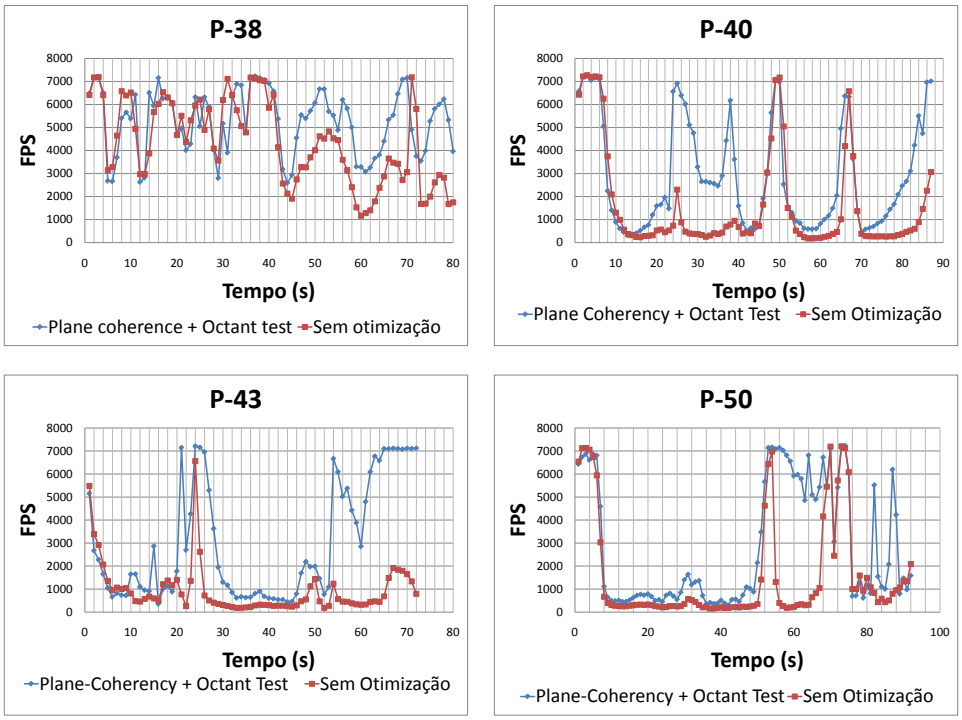


Figura 4.7: Otimizações sugeridas por Assarsoon nas plataformas.

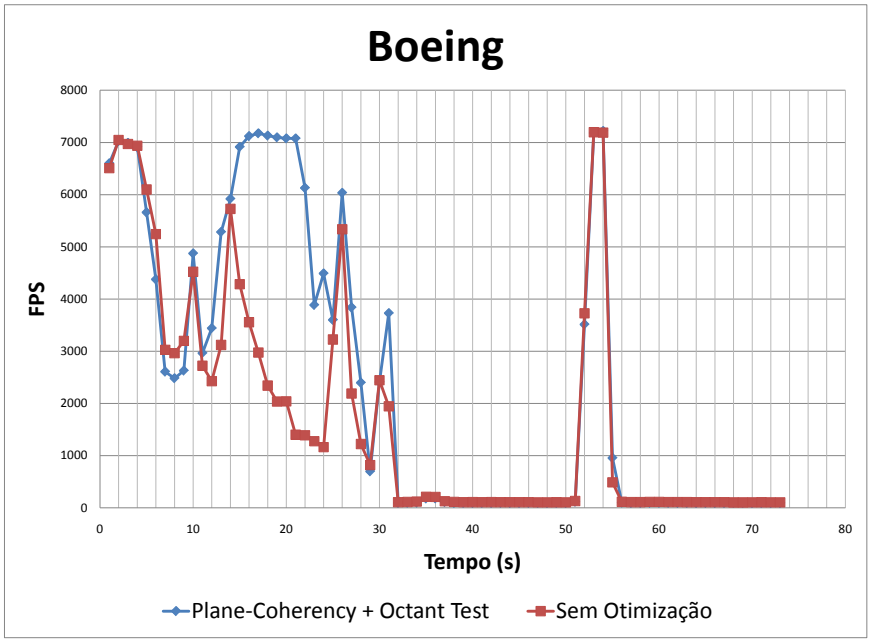


Figura 4.8: Otimizações sugeridas por Assarsoon no Boeing.

testes com as plataformas P-38 e P-50 aumentaram os seus desempenhos com a utilização do teste entre AABBs em 1.58% e 1.28% respectivamente.

As Figuras 4.9 e 4.10 mostram as performances obtidas nos caminhos de câmera com todas e sem nenhum tipo de otimização nas plataformas e no Boeing. Os testes sem otimização foram feitos utilizando apenas hierarquia e testando dois vértices da AABB contra o *frustum*, enquanto os testes com otimização diferem apenas na utilização ou não do teste entre a AABB da câmera e o volume envolvente da geometria e o tipo de hierarquia construída, como pode ser visto na Tabela 4.8.

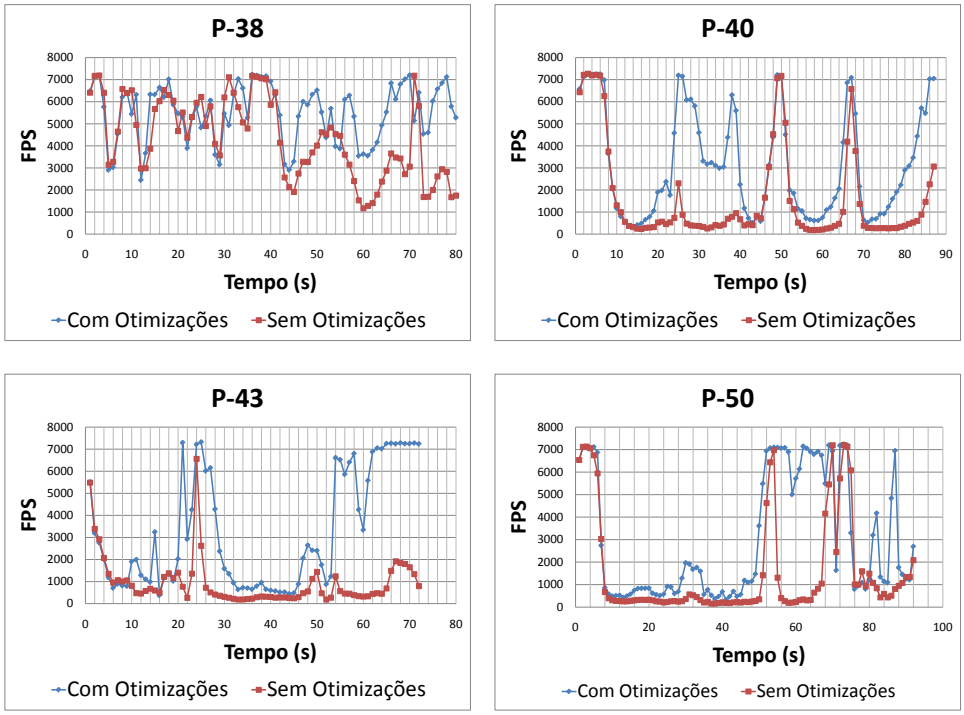


Figura 4.9: Caminhos de câmera pelas plataformas com otimizações.

Em poucos *frames* as otimizações não obtiveram performance superior ao teste de *frustum culling* sem otimização. As causas desse problema podem ser explicadas pela captura destes *frames* em momentos diferentes ou a influência de fatores externos como escalonamento de processos da máquina.

4.6 SIMD

Foram identificados três lugares para a utilização de SIMD no algoritmo de *frustum culling*, sendo cada um deles referente a uma otimização. O primeiro local onde o SIMD pode ser utilizado é na extração dos planos do *frustum*, na abordagem vista na Seção 3.1. Esses cálculos envolvem normalizações que

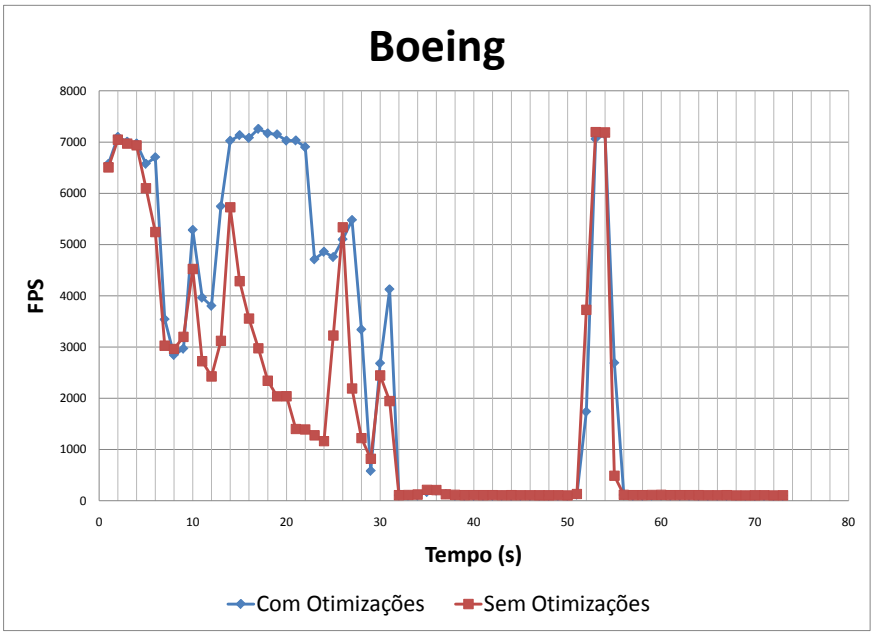


Figura 4.10: Caminhos de câmera pelo Boeing com otimizações.

Modelo	Hierarquia	Planos n e p	Plane-coherency	Octant-test	Ausência de pilha	Teste entre AABBs
P-38	Média	Sim	Sim	Sim	Sim	Sim
P-40	Mediana	Sim	Sim	Sim	Sim	Não
P-43	Mediana	Sim	Sim	Sim	Sim	Não
P-50	Média	Sim	Sim	Sim	Sim	Sim
Boeing	Média	Sim	Sim	Sim	Sim	Não

Tabela 4.8: Melhor combinação de otimizações utilizadas em cada modelo.

podem ser tratadas de maneira paralela. Apesar do gargalo da aplicação não estar nessa parte do código, quando se utiliza modelos massivos, essa técnica foi incorporada ao *pipeline* para ser aproveitada em modelos menores que os tratados nesta dissertação.

Outro local onde é possível tirar proveito do SIMD é nos cálculos de descarte de volumes envolventes, mais especificamente no cálculo de distância. A utilização de SIMD nesta parte do algoritmo não foi de grande valia, uma vez que é necessário construir um novo tipo de 128 *bits* no mínimo para cada teste, o que é mais custoso que fazer o teste diretamente. A Tabela 4.9 compara os caminhos de câmera com e sem SIMD na plataforma P-43.

A construção de no mínimo um novo tipo de 128 *bits* a cada teste feito diminuiu a performance do caminho de câmera da P-43 em 51.66%.

O SIMD também foi utilizado no cálculo da interseção dos planos para construção da AABB do *frustum* que é utilizada nos testes feitos entre AABBs. Essa parte do código segue a mesma ideia da atualização da câmera, pois é

Método	Frames Processados	FPS Mínimo	FPS Máximo	Média do Caminho
Sem SIMD	2234	21	70	37.233
Com SIMD	1473	15	47	24.55

Tabela 4.9: SIMD nos cálculos de descarte da P-43.

feito apenas uma vez a cada *frame* e seu cálculo envolve várias operações.

A conclusão é que a utilização do SIMD só é válida quando há cálculos mais pesados envolvidos que o de descarte de volumes envolventes. Sendo assim, só foram incorporados SIMD na atualização da câmera e no cálculo de sua AABB, porém sem trazer ganho significante para o algoritmo de *frustum culling* nos modelos em questão.

4.7 Multiprocessamento

Como sistemas multiprocessados estão ficando cada vez mais populares, muitas aplicações têm sido desenvolvidas em cima desses sistemas. Um sistema multiprocessado possui dois ou mais processadores físicos, onde cada um é capaz de executar processos simultâneos automaticamente compartilhando um único espaço de memória. O desenvolvimento da aplicação *multithread* foi feito com a biblioteca OpenMP [52], que é uma API para o desenvolvimento de aplicações *multithread* em C, C++ e Fortran disponível em vários sistemas operacionais.

A ideia da utilização de *multithread* tenta reduzir o gargalo da aplicação nos cálculos de descarte. O grande problema enfrentado por Assarsson e Stenstrom *et al.* [4] foi encontrar o melhor balanceamento das tarefas entre os processadores. Assarsson utilizou quatro abordagens diferentes a fim de encontrar o melhor balanceamento fazendo o teste em três tipos de cenas. Mesmo tendo poder de processamento e cenas muito menores do que as utilizadas neste trabalho, foram tomadas como base as duas maiores cenas onde a abordagem chamada de *Lock-free Scheme* obteve melhor desempenho, como pode ser visto na Figura 4.11.

Na cena de tamanho médio em alguns momentos a distribuição *Hybrid Scheme* obtém melhor desempenho que o *Lock-free Scheme*, o que não ocorre na cena maior, onde sempre que o número de processadores é aumentado, o desempenho melhora a uma velocidade maior que as outras distribuições. A distribuição de tarefas *Lock-free Scheme*, desenvolvida por Assarsson, foi implementada com o auxílio da biblioteca OpenMP versão 2.5 (2005). Além dessa abordagem, foram desenvolvidas mais duas para servirem como base de comparação. A primeira não compartilha tarefas entre os processadores, sendo

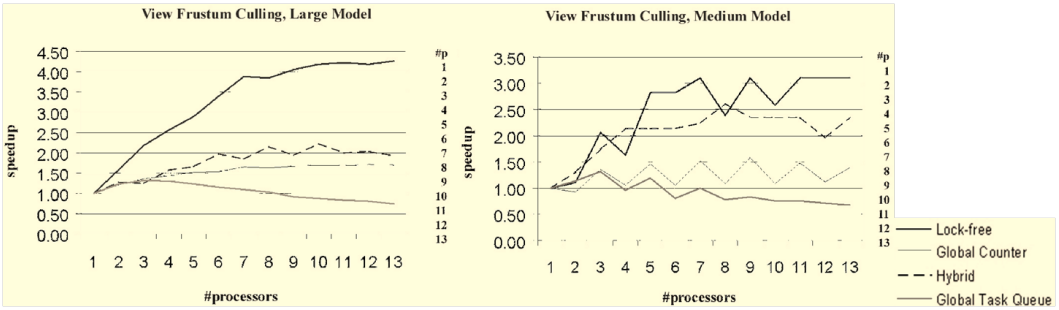
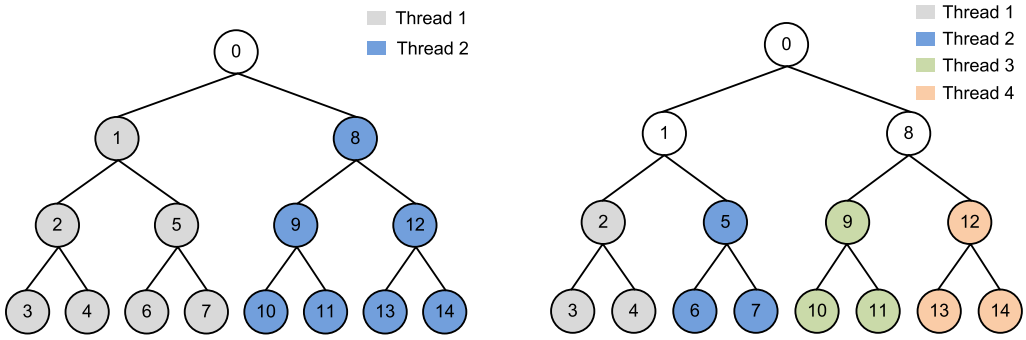


Figura 4.11: Resultados obtidos por Assarsson em duas cenas.

assim, quando as tarefas acabam para um processador ele fica aguardando a tarefa dos outros terminar. A Figura 4.12 ilustra a distribuição de tarefas para dois 4.12(a) e quatro processadores 4.12(b), onde os nós de mesma cor irão ser processados pelas *threads* correspondentes à sua cor.



4.12(a): Percurso com duas threads.

4.12(b): Percurso com quatro threads.

Figura 4.12: Percurso com *multithread*.

A segunda abordagem tenta explorar a versão 3.0 (2008) da biblioteca OpenMP. Esta versão introduziu a ideia de *tasks*. *Tasks* são tarefas desenvolvidas dentro das *threads* que podem ser suspensas ou interrompidas. A grande vantagem da utilização de *tasks* ocorre quando dentro de uma *thread* todos as suas tarefas terminam. Neste ponto suas *tasks* podem ser compartilhadas com as outras *threads*, o que não era possível na versão 2.5 da biblioteca OpenMP. No caso do percurso de hierarquia com *multithread* utilizando *tasks*, a distribuição de tarefas fica implícita não havendo necessidade de troca de dados entre *threads*.

As Figuras 4.13 e 4.14 ilustram a comparação de desempenho obtido com o percurso da hierarquia utilizando uma, duas e quatro *threads* nas plataformas e no Boeing.

Nos testes realizados, apenas na plataforma P-50 com 2 *threads* houve ganho no percurso da hierarquia sem o compartilhamento de tarefas entre *threads*. Em todos os outros modelos a utilização de apenas uma *thread* no

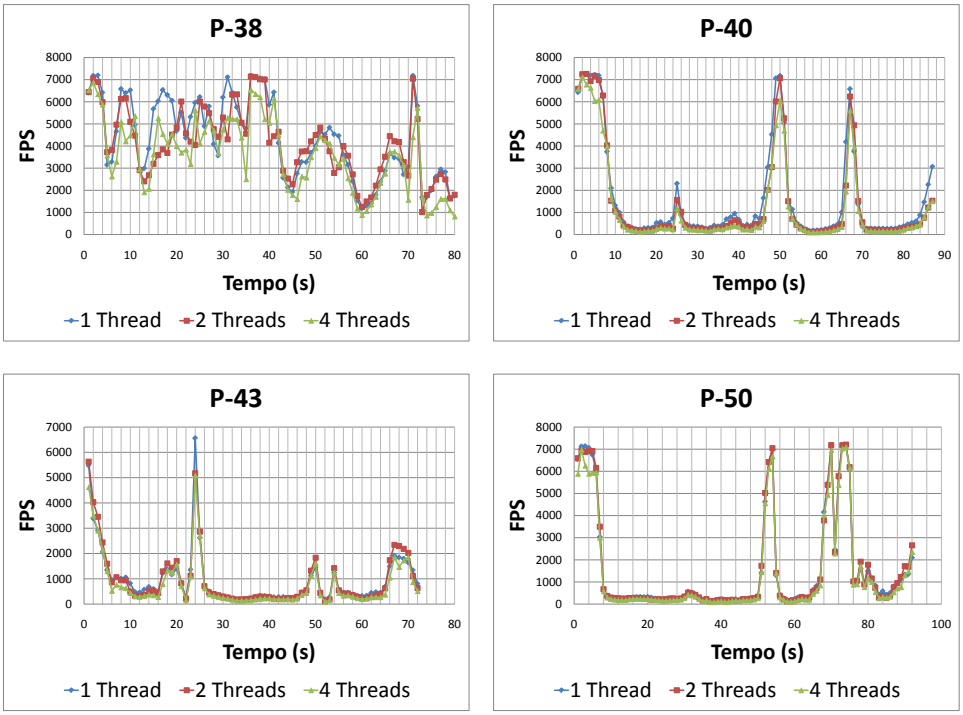


Figura 4.13: Multiprocessamento sem troca de tarefas entre threads.

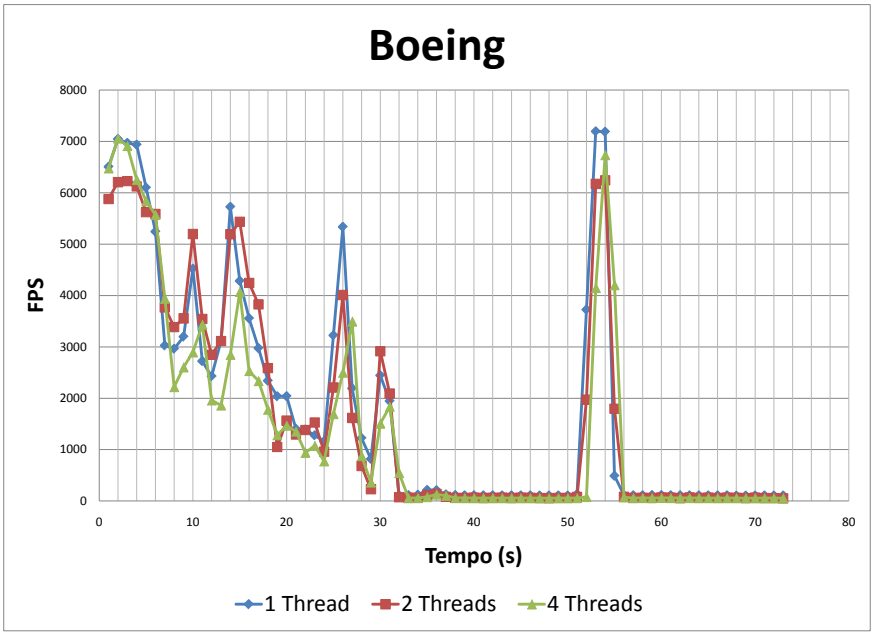


Figura 4.14: Multiprocessamento sem troca de tarefas entre threads no Boeing.

percurso obteve melhor performance. Quanto mais *threads* foram colocadas para realizar o percurso, menor o desempenho.

As Figuras 4.15 e 4.16 ilustram a performance dos algoritmos que trocam tarefas entre as *threads*. Os dois algoritmos implementados, *Lock-Free Scheme* e *Tasks*, utilizaram apenas duas *threads*.

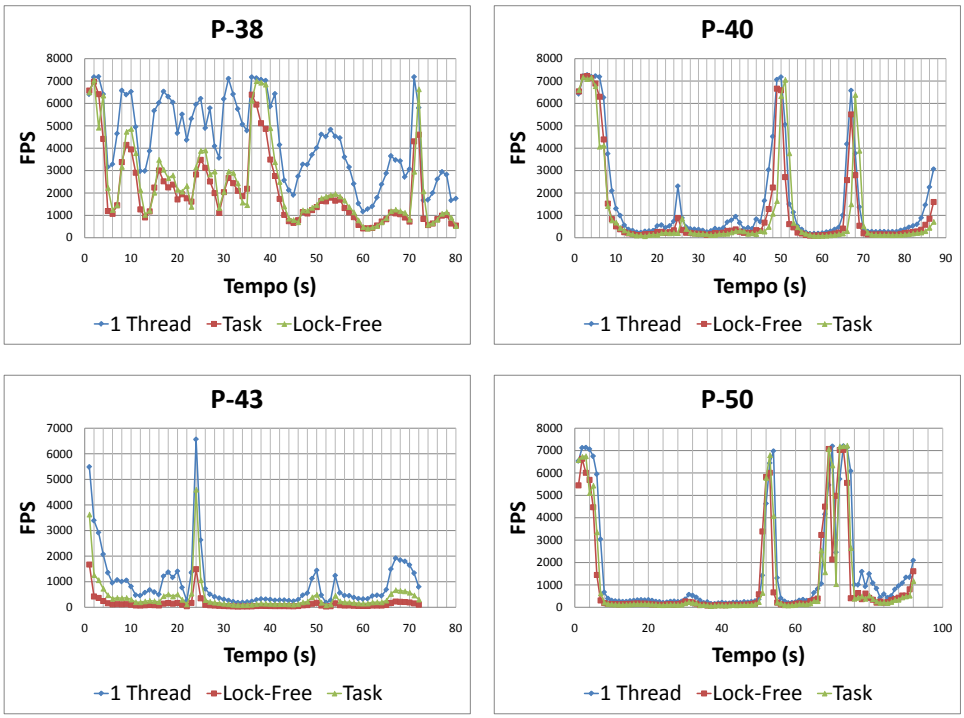


Figura 4.15: Multiprocessamento com troca de tarefas entre threads nas plataformas.

Os resultados obtidos com a implementação do algoritmo *Lock-free Scheme* não foram satisfatórios. Foram encontradas dificuldades de implementação que não estão bem explicadas no trabalho de Assarsson, como por exemplo uma métrica para distribuição dos nós. Outro fator que pode ter influenciado é a quantidade de processadores. Neste trabalho foram utilizados quatro e no trabalho de Assarsson quatorze.

A implementação de *tasks* não obteve o desempenho esperado, uma vez que em teoria poderia obter melhor performance. O baixo desempenho dessa abordagem pode ser explicado por configurações do compilador utilizado e o número de *tasks* a serem criadas.

A conclusão sobre o multiprocessamento é que a princípio não vale a pena o custo de distribuir as tarefas de percurso da hierarquia por mais de uma *thread* e muito menos compartilhá-las entre as *threads* utilizando a biblioteca OpenMP.

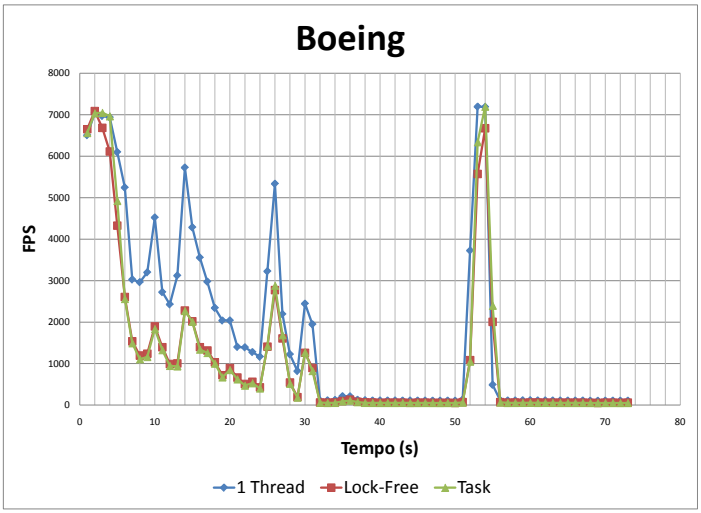


Figura 4.16: Multiprocessamento com troca de tarefas entre threads no Boeing.

4.8 Pipeline final em CPU

O esquema do melhor algoritmo de *frustum culling* implementado em CPU para a maioria dos modelos, pode ser visto na Figura 4.17.

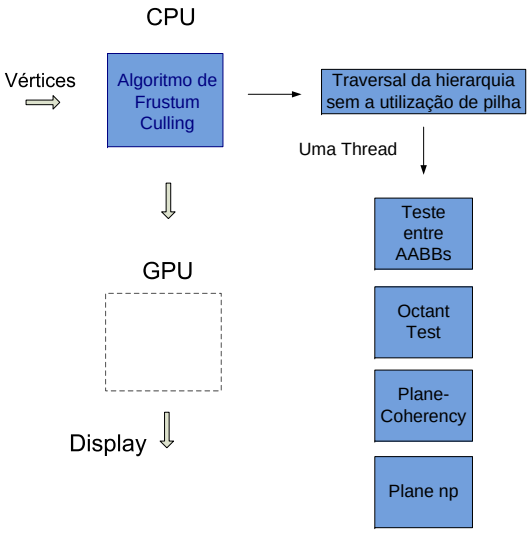


Figura 4.17: Pipeline final em CPU.