Universidade Federal do Pampa

Autor: Rafaelo Pinheiro da Rosa

ANÁLISE DE SENSIBILIDADE DA VARIAÇÃO QUANTITATIVA DE MÉTRICAS DE QUALIDADE SOBRE A EFICIÊNCIA DE UM RAYTRACER

Trabalho de Conclusão de Curso II

BAGÉ
2012

RAFAELO PINHEIRO DA ROSA

ANÁLISE DE SENSIBILIDADE DA VARIAÇÃO QUANTITATIVA DE MÉTRICAS DE QUALIDADE SOBRE A EFICIÊNCIA DE UM RAYTRACER

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Computação.

Orientador: Bruno Silveira Neves

Bagé
2012

RAFAELO PINHEIRO DA ROSA

ANÁLISE DE SENSIBILIDADE DA VARIAÇÃO QUANTITATIVA DE MÉTRICAS DE QUALIDADE SOBRE A EFICIÊNCIA DE UM RAYTRACER

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Computação.

Trabalho de Conclusão de Curso defendido e aprovado em: 20 de novembro de 2012
Banca examinadora:

_____

Prof. MSc. Bruno Silveira Neves
Orientador
UNIPAMPA

_____

Prof. Dr. Leonardo Bidese de Pinho
Coorientador
UNIPAMPA

_____

Prof. MSc. Carlos Michel Betemps
UNIPAMPA

Dedico este trabalho aos meus amados avós, Alzi e Altair, maiores incentivadores e fontes inesgotáveis de apoio, amor e compreensão.

AGRADECIMENTO

Aos meus pais José Carlos e Edna de Fátima pelo amor e educação fornecidos para que eu pudesse me tornar a pessoa que sou hoje.

A minha namorada Bruna Fonseca por todo apoio e incentivo dado a mim durante estes anos de curso.

Ao Prof. MSc. Bruno Silveira Neves pela dedicação e paciência durante os quase três anos de orientação.

Ao Prof. Dr. Leonardo Bidese de Pinho por ter me auxiliado em determinadas etapas deste trabalho.

A todos os colegas de curso pelo convívio e pelos momentos de amizade.

A todas as pessoas que, direta ou indiretamente contribuiram para a realização desta pesquisa.

O único lugar onde o sucesso vem antes do trabalho é no dicionário.

Albert Einstein

RESUMO

Atualmente, a fabricação em grande escala de sistemas multiprocessados tem sido acompanhada por uma revisão das técnicas para desenvolvimento de softwares paralelos. Contudo, percebe-se que a forma com a qual os aplicativos paralelos estão sendo desenvolvidos está orientada, prioritariamente, para a redução do tempo de execução, sendo desconsideradas, deste modo, métricas tradicionais de qualidade de software como reusabilidade, manutenibilidade e escalabilidade. Além disso, outras métricas relacionadas à eficiência como consumo energético e de memória, importantes sobretudo em áreas emergentes no campo do desenvolvimento de software como a de sistemas embarcados, também vem sendo negligenciadas. Assim sendo, o objetivo deste trabalho é demonstrar um relacionamento entre métricas de qualidade de software, favoráveis ao reuso de software, manutenibilidade e redução do tempo de execução, e métricas de eficiência, tais como ciclos gastos, energia, consumo de memória, entre outras, buscando, assim, identificar os impactos que a variação das métricas de softwares causam sobre a eficiência de uma aplicação paralela quando executada em uma plataforma multiprocessada embarcada.

Palavras-chave: paralelismo, métricas de qualidade, software.

ABSTRACT

Currently, large-scale manufacture of multiprocessor systems has been accompanied by a review of techniques for parallel software development. However, it is clear that the way in which parallel applications are being developed is oriented primarily to reduce the execution time, being disregarded, thus traditional metrics of software quality such reusability, maintainability and scalability. In addition, other metrics related to efficiency such energy consumption and memory, especially in important emerging areas in the field of software development such as embedded systems, has also been neglected. Therefore, the objective of this study is to demonstrate a relationship between software quality metrics, in favor of software reuse, maintainability and reducing the execution time and efficiency metrics such as cycles, energy, memory consumption, among others, thereby potentially identify impacts that the change in software metrics have on the efficiency of a parallel application when run on an embedded multiprocessor platform.

Keywords: parallelism, quality metrics, software.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Atualmente, processadores de um só núcleo não conseguem mais oferecer a capacidade computacional demandada por grande parte do software existente, incluindo sistemas operacionais e aplicativos multimídia como jogos, processadores de imagens e vídeo, além de outros tipos de aplicações como CAD (*Computer Aided Design*) para engenharia, entre outros (CARDOSO; ROSA; FERNANDES, 2011). A solução encontrada, até recentemente, pelos desenvolvedores de *hardware*, era de aumentar a frequência de operação dos processadores de propósito geral com um único núcleo, o que permitiria uma possível redução no tempo de execução gerando um melhor desempenho para o processamento das aplicações citadas acima. Para que houvesse este aumento na frequência de operação, uma das alternativas usadas explorava o projeto diretamente no nível de transistores, buscando com que estes fossem fabricados em tamanhos cada vez menores. Alternativamente, ajustes na arquitetura e organização do *hardware*, baseadas em técnicas como paralelismo e *pipeline* também possibilitavam maior frequência de operação sem mexer na tecnologia do transistor. Entretanto, segundo (ZHIRNOV et al., 2003), essas alternativas para aumento de desempenho das aplicações atualmente tem sofrido limitações devido a aspectos como perda da confiabilidade e aumento do consumo energético do processador, decorrentes da elevada frequência de operação (que incrementa o aquecimento dos transistores) e correntes de fuga produzidas pela redução da área dos dispositivos.

A solução encontrada para contornar este problema passou a ser então o aumento do número de núcleos dentro de um único *chip*, resultando no desenvolvimento de sistemas multiprocessados. Tais sistemas estão sendo fabricados em grandes quantidades ultimamente, tendo como objetivo suprir a crescente necessidade computacional demandada pelas aplicações existentes atualmente. Uma das empresas que está atuando no ramo de fabricação de processadores *multicore* é a Intel (INTEL, 2011), com a produção de *chips* como o Core 2 Duo, Core 2 Quad, e outros mais recentes como o Core i7. Outra empresa que ocupa lugar nesse mercado de sistemas multiprocessados é a AMD (AMD, 2011), com a fabricação de processadores como Athlon 64, Turion 64 X2, Phenom II, entre outros. Sistemas com múltiplos processadores levam algumas vantagens em relação a sistemas *single core*. Entre as principais vantagens pode-se citar um melhor desempenho para aplicações *multithreading* e desempenho superior em cenário de execução de múltiplos aplicativos que utilizam processamento de forma intensiva (CARDOSO; ROSA; FERNANDES, 2011).

Na tentativa de acompanhar esta crescente tendência pelo desenvolvimento de multiprocessadores, ganham ênfase os padrões de projeto voltados para a programação paralela como meio de explorar, o máximo possível, os recursos oferecidos pelo *hardware* moderno. Padrões de

projeto (DEITEL; DEITEL, 2006) podem ser definidos como estruturas já testadas anteriormente servindo como base para o desenvolvimento de programas orientados a objetos. Estes padrões permitem gerenciar adequadamente a complexidade inerente ao desenvolvimento de uma aplicação auxiliando na tarefa de identificar a melhor forma para sua implementação sob o ponto de vista do desempenho.

Entretanto, observa-se que os avanços no campo de padrões de projeto para software paralelo está voltada, prioritariamente, para a redução do tempo de execução da aplicação, deixando de lado aspectos qualitativos de software como reusabilidade, manutenibilidade e escalabilidade. Além disso, no que diz respeito à eficiência de execução, outras métricas importantes, sobretudo para aplicações que executam sobre multiprocessadores embarcados, tais como consumo energético e de memória, não vêm sendo consideradas pelos desenvolvedores durante o aprimoramento destes padrões (MILLS, 1998).

Desta forma, o objetivo deste trabalho é relacionar as métricas de qualidade de software (METRICS, 2011) (XENOS et al., 2000) e as métricas de eficiência para que seja possível analisar os impactos causados, em alto nível de abstração, pela variação de um conjunto de métricas de software, sobre as métricas de eficiências inerentes à execução de uma aplicação embarcada paralela, dando sequência ao trabalho desenvolvido anteriormente, explicado detalhadamente no Capítulo 4. Isso contribui para que os projetistas sejam orientados durante o desenvolvimento de softwares paralelos. Adicionalmente, a metodologia proposta constitui um ferramental para que possam ser desenvolvidas uma ou mais versões de um determinado aplicativo, sem alterar o grau de paralelismo, buscando fazer com que cada versão dê ênfase maior a um dos aspectos qualitativos do projeto de software (com base em um conjunto de aspectos selecionados) para, então, coletar as métricas de software e as métricas de eficiência, incluindo o consumo de memória destas versões desenvolvidas, com o objetivo de identificar e justificar os impactos causados pela variação das métricas de software sobre as métricas de eficiência.

1.1 Estrutura do Trabalho

Este trabalho esta dividido da seguinte forma: no Capítulo 2 é apresentada a definição de algumas métricas de software utilizadas neste TCC. No Capítulo 3 segue uma descrição detalhada da aplicação Ray Tracing, apresentando os principais conceitos das técnicas utilizadas por esta aplicação. O Capítulo 4 apresenta os experimentos desenvolvidos e os resultados obtidos anteriormente à este TCC, apresentando, logo em seguida, a metodologia proposta para o desenvolvimento deste trabalho bem como as etapas já concluídas. Este capítulo apresenta também, nas seções iniciais, uma descrição breve do processador FemtoJava (ITO, 2000) e das

ferramentas de instrumentação utilizadas, como a BIT (LEE; ZORN, 2011) e a CAT (NEVES, 2005).

2 MÉTRICAS DE SOFTWARE ORIENTADA A OBJETOS

Para que um projeto de software possa ser analisado e avaliado, é necessário que ele passe por um tipo de medição (WIKIPEDIA, 2011). Métricas de Software permitem verificar, através de dados quantitativos, a qualidade que um determinado projeto de software, em desenvolvimento, possui. Desta forma, estas medições possibilitam alertar, precocemente, os problemas em potencial que surgem no decorrer do processo de software (SCHACH, 2009). Por exemplo, uma métrica como tempo médio entre defeitos poderia fornecer aos desenvolvedores de software um nível de confiabilidade de um produto recentemente instalado nos computadores de um cliente. Se um determinado produto começar a falhar uma vez sim e outra vez não, sua qualidade está abaixo em comparação com um produto similar que, em média, executa por um bom período de tempo sem ocorrer defeitos.

Várias métricas de software estão diretamente relacionadas com a qualidade de software. Pode-se considerar que as métricas de software se aplicam a maioria das linguagens de programação. Entretanto, existe um conjunto de métricas de software que se aplicam somente às linguagens orientadas a objetos (OO) (XENOS et al., 2000). O paradigma de programação orientado a objetos é uma estratégia utilizada para planejar, analisar e desenvolver projetos de software através da estruturação e do relacionamento de unidades mínimas de software, denominados de objetos. As métricas de software orientadas a objetos são as responsáveis pela medição de programas orientados a objetos, buscando mensurar o quão bem codificados estão os programas escritos no escopo deste paradigma.

2.1 Métricas OO utilizadas neste trabalho

Esta subseção irá apresentar uma breve descrição sobre as métricas OO utilizadas neste TCC para que seja permitido verificar o impacto que cada uma delas causa na eficiência de uma aplicação.

Abstração (A do inglês *Abstractness*): esta métrica permite analisar a capacidade que uma classe tem em ser estendida (OLIVEIRA et al., 2011). Classes abstratas e interfaces possuem algumas características que podem ser mais facilmente herdadas, ou serem estendidas, para um conjunto de classes concretas (classes que podem serem usadas para instanciar objetos (DEITEL; DEITEL, 2006)), favorecendo assim o reuso de software. A medição desta métrica é feita através da divisão do número total de classes abstratas e interfaces pelo número total de classes concretas em um pacote. Estes pacotes são pastas ou diretórios de um sistema operacional sendo utilizados para agrupar classes semelhantes (TI, 2011). Se o valor gerado por esta divisão

for alto, então isso significa que mais componentes estão sendo estendidos. Caso contrário, as classes deste projeto estão mais próximas de serem classes concretas.

Profundidade de Herança (DIT do inglês *Depth of Inheritance Tree*): esta métrica determina a distância encontrada entre a classe analisada em relação às suas classes ancestrais. Quanto maior for esta distância, maior será o número de componentes a serem herdados (OLIVEIRA et al., 2011). Isto pode ser visto na hierarquia à esquerda da Figura 1, onde a classe D poderia herdar os componentes da classe C, assim como poderia herdar, também, os componentes das classes A e B. Esta maior distância na hierarquia tornaria o projeto mais complexo, uma vez que o número de componentes a serem herdados por D aumentam. Entretanto, pode favorecer a reutilização de software, uma vez que uma determinada classe poderia utilizar um método implementado em sua superclasse. Em contra partida, uma distância menor torna a solução do projeto mais simples, embora acabe diminuindo a capacidade de reutilização. A Figura 1 mostra a distância encontrada entre as classes C e E em relação à classe ancestral A, em duas árvores de herança. Nesta figura, a hierarquia da esquerda apresenta uma melhor reutilização de software em comparação com a da direita, uma vez que os componentes herdados pela classe D, em relação a superclasse A, são maiores que os componentes herdados por D, também em relação a classe A da hierarquia apresentada a direita.

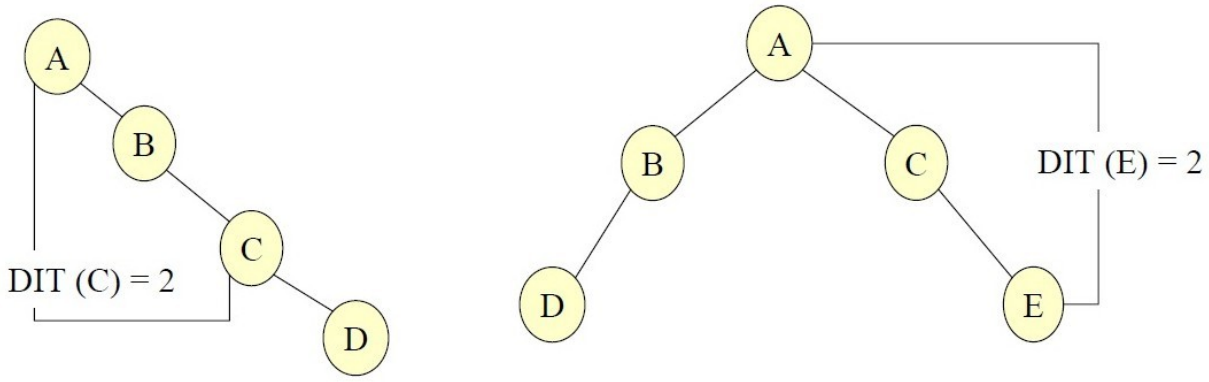

Figura 1: Cálculo da distância na profundidade de herança. Fonte: (MARTINS, 2003)

Falta de Coesão nos Métodos (LCOM do inglês *Lack Cohesion of Methods*): em projetos orientados a objetos, torna-se desejável a existência de alta coesão entre os componentes que formam a classe, uma vez que permite um melhor grau de modularidade. Uma classe coesa indica que seus métodos estão bem definidos, com funcionalidades mais próximas das características específicas apresentadas pelo objeto instanciado por esta classe, havendo poucos acessos a outros componentes pertencentes a objetos externos. Dois métodos de uma classe C são coesos se, e somente se, estes métodos compartilham atributos desta mesma classe. Esta métrica faz suas medições com a utilização do método apresentado por Henderson-Sellers (HENDERSON-

SELLERS, 1996), onde *m(Aj)* é uma função que descreve o número de métodos que acessam cada atributo *A* de uma determinada classe. O cálculo é feito através de uma média de acessos para todos os atributos da classe, subtraindo-se este valor do número de métodos *m* e dividindo-se o resultado por *(1 - m)*. Este cálculo é mostrado na equação 2.2 (LCOM, 2011). Valores próximos de 0 indicam que a classe está coesa. Valores perto de 1 indicam uma falta de coesão. Sugere-se, neste último caso, que a classe seja dividida em subclasses, como alternativa para melhorar a coesão (OLIVEIRA et al., 2011).

$$\mu = \frac{1}{a} \sum_{j=1}^{a} m(Aj) \tag{2.1}$$

$$LCOM = \frac{\mu - m}{1 - m} \tag{2.2}$$

Onde

m = número de métodos.
a = número de atributos.
$\{Aj\}(j = 1, 2, ..., a)$ = conjunto de atributos.
$m(Aj)$ = número de métodos que acessam cada atributo.

Número de Métodos (NOM do inglês *Number of Methods*): esta métrica representa o número total de métodos que a classe analisada possui. Medições com altos valores indicam uma maior modularização (OLIVEIRA et al., 2011), porém pode significar também mais acessos a outros métodos, o que pode gerar uma perda em termos de desempenho, uma vez que existiriam mais instruções para chamadas e retornos de métodos.

Acoplamento Aferente (Ca do inglês *Afferent Coupling*): esta métrica representa o relacionamento do número de classes externas a um pacote com o número de classes presentes internamente a este mesmo pacote (OLIVEIRA et al., 2011).

Acoplamento Eferente (Ce do inglês *Efferent Coupling*): esta métrica representa o relacionamento do número de classes internas a um pacote com o número de classes presentes externamente a este mesmo pacote (OLIVEIRA et al., 2011).

Número de Parâmetros (NOP do inglês *Number of Parameters*): esta métrica indica o número médio de parâmetros que a classe alvo possui. Quanto mais informações forem trocadas entre os métodos de uma classe analisada, mais facilmente será a leitura e a escrita desses tipos de variáveis, devido à sua alocação em uma pilha de dados (OLIVEIRA et al., 2011). Os valores gerados por esta métrica são determinados a partir da divisão do número total de parâmetros

pelo número total de métodos que a classe alvo possui.

Número de Atributos Estáticos (NOSA do inglês *Number of Static Attributes*): esta métrica indica o número total de atributos estáticos presentes na classe analisada. Altos valores gerados por esta medição podem levar a um melhor desempenho no tempo de execução (OLIVEIRA et al., 2011). Em contra partida, podem significar um maior consumo de memória quando comparados a objetos dinâmicos, uma vez que os dados estáticos tendem a permanecer alocados na memória durante toda a execução da aplicação.

Número de Métodos Estáticos (NOSM do inglês *Number of Static Methods*): esta métrica representa o número total de métodos estáticos presentes na classe alvo. Assim como na métrica NOSA, valores altos gerados pelo NOSM podem indicar um bom desempenho devido ao fato de chamadas a métodos estáticos serem mais rápidas que métodos dinâmicos (OLIVEIRA et al., 2011). Um ponto negativo encontrado nestes valores altos é que esta solução para o desenvolvimento da aplicação levaria a uma diminuição na capacidade de reutilização de software.

3 RAY TRACING

A área da computação responsável pela criação de imagens que simulam o foto-realismo é a computação gráfica. Esta área divide-se em dois ramos, basicamente: a modelagem, responsável pela criação de imagens utilizando ferramentas matemáticas, e a visualização, que possui técnicas para a representação gráfica da imagem real (SILVA, 2011). Uma das técnicas de visualização, muito utilizada devido a sua importância, é o Ray Tracing.

Esta técnica é um algoritmo de computação gráfica que realiza a renderização de imagens tridimensionais. Renderização é uma alternativa para fazer a conversão de um tipo de arquivo em outro (SALES, 2008). Este algoritmo utiliza um método que busca simular, de forma invertida, a trajetória dos raios luminosos na natureza. No mundo real, uma fonte de luz gera raios luminosos que tracejam pelo espaço até se intersectar com um objeto, permitindo que suas trajetórias sejam desviadas, ou seja, sejam refletidas ou refratadas, devido às características físicas deste objeto, como transparência, coloração e textura, possibilitando que apenas uma pequena minoria dos raios com as trajetórias desviadas estejam ao alcance dos olhos do observador.

O algoritmo Ray Tracing inverte a simulação da trajetória convencional dos raios luminosos, permitindo que a fonte de luz passe a ser agora os olhos do observador. Se o algoritmo utilizasse a trajetória real, a quantidade de processamento gasto para simular esta trajetória seria praticamente inviável devido a pequena minoria dos raios que atingem os olhos do observador, uma vez que esta minoria de raios são fundamentais para o processamento, em comparação com os raios emitidos pela fonte de luz.

Este capítulo trata da descrição da aplicação escolhida para a realização dos estudos deste trabalho. O Ray Tracing foi escolhido por ser uma aplicação paralela utilizado em dispositivos embarcados, como celulares e video-*games*, uma vez que o objetivo deste trabalho é analisar os impactos que as modificações das métricas OO causam na eficiência de uma aplicação embarcada paralela. Na sequência deste capítulo, serão abordadas outras seções, cujo foco será, inicialmente, a apresentação dos conceitos fundamentais do Ray Tracing, seguida da introdução de alguns efeitos produzidos por este algoritmo, como reflexão, refração e sombra.

3.1 Conceitos básicos do Ray Tracing

O Ray Tracing é o algoritmo mais simples dentre todos existentes para a visualização de objetos (SILVA, 2011). Sua técnica baseia-se na seguinte forma: coloca-se uma tela, com transparência, para atuar como interface entre o observador e os objetos tridimensionais, que foram

construídos com a utilização de alguma técnica de modelagem. Neste cenário, a fonte de luz será os olhos do observador que emitirá raios visuais que cruzarão cada um dos pontos da tela transparente, mantendo sua trajetória até atingirem um objeto. No momento em que o objeto for encontrado, o ponto da tela que foi atravessado pelo raio será pintado com a cor do objeto atingido por este raio.

O Ray Tracing possui um algoritmo fundamental que é mostrado a seguir:

Para cada ponto na tela transparente, faça:

Crie um raio que una os olhos do observador com um dos pontos na tela.
Descobrir se este raio atingiu algum objeto tridimensional.
Determinar a cor do objeto atingido neste ponto.
Pintar o ponto com esta cor.

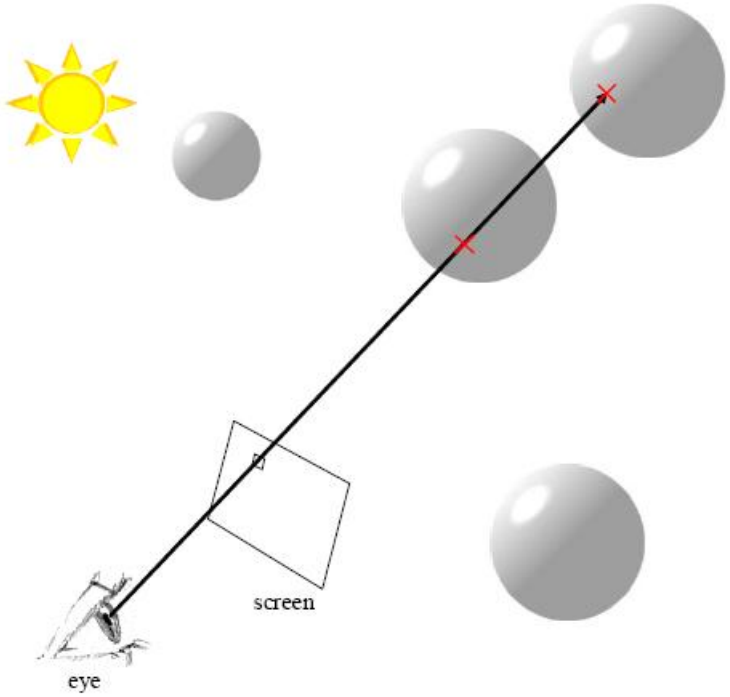

Figura 2: Representação de um raio tracejado pelo algoritmo Ray Tracing. Fonte: (SILVA, 2011)

A Figura 2 mostra um raio sendo projetado dos olhos do observador, atravessando um ponto na tela e atingindo, logo em seguida, um objeto tridimensional. Estas interseções entre os raios visuais, emitidos pelo observador, e os objetos tridimensionais poderiam ser consideradas como o *overhead* do algoritmo, uma vez que esta etapa utiliza 75% a 95% de tempo para o cálculo das interseções entre os raios e os objetos, mostrando que esta rotina pode afetar diretamente

no seu desempenho (WHITTED, 1980).

Esta seção apresenta brevemente uma versão simplificada do Ray Tracing e como ocorre a interseção de raios em objetos na cena.

3.1.1 Ray Casting

Este algoritmo é uma versão mais simples do Ray Tracing sendo utilizado para a visualização volumétrica de objetos tridimensionais. Sua técnica é praticamente a mesma apresentada pelo Ray Tracing: raios visuais são emitidos pelos olhos do observador de forma a perceber a distância entre cada um dos objetos que estão representados na cena (SILVA, 2011). Esta técnica permite uma melhor visualização no detalhamento interno do objeto, podendo remover ou ignorar as superfícies escondidas em uma imagem.

A Figura 3 apresenta imagens que são visualizadas pelo Ray Casting.

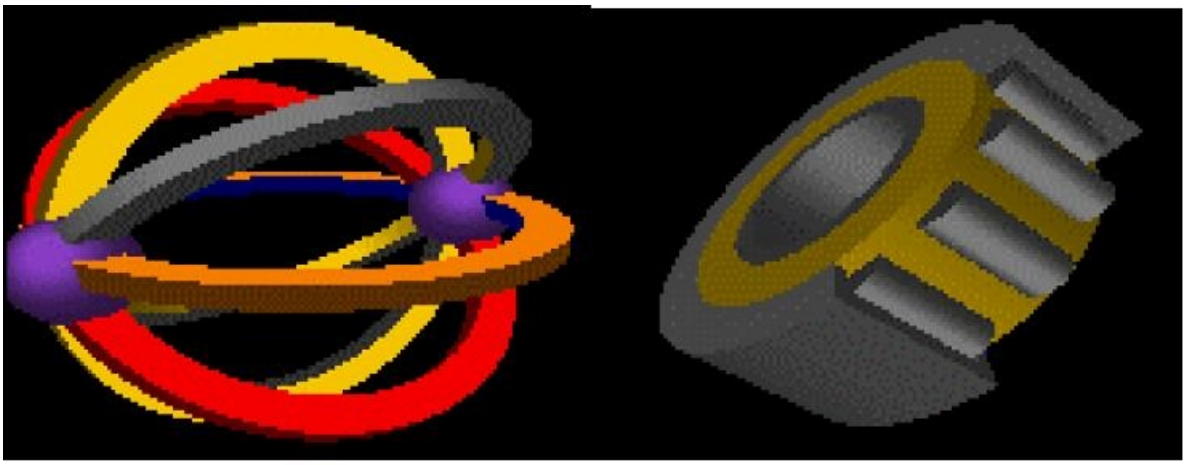

Figura 3: Representação de objetos utilizando o Ray Casting. Fonte: (SILVA, 2011)

3.1.2 Cálculo de interseções

A determinação da interseção do raio com o objeto na cena é uma etapa importante no algoritmo Ray Tracing. Nesta etapa, os raios são representados por uma linha reta no espaço tridimensional do observador, sendo descritos com os mesmos parâmetros de um vetor (x, y, z) (FOLEY, 1993). Estes raios vão de um ponto inicial $(x_0,\ y_0,\ z_0)$ até um ponto final $(x_1,\ y_1,\ z_1)$, sendo acessados por um parâmetro t, descrito tal como:

$$x = x_0 + t(x_1 - x_0), \quad y = y_0 + t(y_1 - y_0), \quad z = z_0 + t(z_1 - z_0). \tag{3.1}$$

Para simplificar, define-se um vetor diretor com parâmetros (Dx, Dy, Dz), onde Dx, Dy e Dz representam

$$Dx = (x_1 - x_0), \quad Dy = (y_1 - y_0), \quad Dz = (z_1 - z_0). \tag{3.2}$$

Assim, a Equação 3.1 fica representada por

$$x = x_0 + tDx, \quad y = y_0 + tDy, \quad z = z_0 + tDz. \tag{3.3}$$

Para o cálculo da interseção do raio com um objeto, é necessário conhecer a equação que representa a superfície deste objeto. Cada tipo de objeto possui uma representação que permite calcular o valor de t durante a interseção deste objeto com o raio. Por exemplo, a esfera, com centro (a, b, c) e raio r, é representada pela Equação 3.4

$$(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2. \tag{3.4}$$

A interseção é encontrada primeiramente expandindo-se a Equação 3.4, com a utilização do teorema do Binômio de Newton.

$$x^2 - 2ax + a^2 + y^2 - 2by + b^2 + z^2 - 2cz + c^2 = r^2, \tag{3.5}$$

Substituindo os valores de x, y e z da Equação 3.3 na Equação 3.5, fazendo novamente a expansão de termos, utilizando o teorema do Binômio de Newton e agrupando os termos semelhantes, a equação resultante fica

$$At^2 + Bt + C = 0 \tag{3.6}$$

onde

$$A = (Dx^2 + Dy^2 + Dz^2) \tag{3.7}$$

$$B = 2[Dx(x_0 - a) + Dy(y_0 - b) + Dz(z_0 - c)] \tag{3.8}$$

$$C = (x_0 - a)^2 + (y_0 - b)^2 + (z_0 - c)^2 - r^2 \tag{3.9}$$

A Equação 3.6 é uma equação de segundo grau em t, sendo resolvida com a fórmula de Báskara. A partir das raízes calculadas utilizando a fórmula de Báskara, pode-se concluir que:

- se não existirem raízes reais, então o raio não atingiu a esfera;
- se existir uma única raíz real, então o raio atingiu um único ponto da esfera, isto é, o raio é tangente à esfera;
- se existirem duas raízes reais, então o raio atingiu a esfera em dois pontos;

O algoritmo que melhor representa este cálculo para a interseção do raio com o objeto pode ser descrito da seguinte forma (SILVA, 2011):

Para cada superfície do sólido faça:
    Resolver a equação que representa a interseção do raio na superfície.
    Se o raio atingir algum ponto na superfície então:
        Armazenar os parâmetros do raio que interseptou a superfície.

Na maioria das vezes, os raios não conseguem atingir nenhum objeto presente na cena. Mesmo assim, é feito todo o processamento do cálculo da interseção do raio com a superfície do objeto. Este tempo de processamento poderia ser considerado como *overhead* uma vez que são feitos cálculos desnecessários para os raios que não interseptam os objetos. Uma forma de contornar este problema é a utilização de sólidos limitantes, como esferas ou blocos. Esta ideia baseia-se em verificar se um raio atingiu um sólido limitante antes dele interseptar a superfície do objeto. Se o raio não atingir o sólido limitante, então ele também não atingirá nenhum outro objeto presente na cena. Desta forma, elimina-se o cálculo de interseção deste raio.

3.2 Ray Tracing recursivo

Este algoritmo utiliza recursividade para tratar dos efeitos como sombra, reflexão e transparência, servindo para determinar a cor do *pixel* mais próximo da interseção do raio visual, lançado pelo observador no objeto, representado por algum modelo de iluminação descrito na próxima seção.

3.2.1 Sombra

Este efeito surge quando existe um obstáculo entre o objeto e a fonte de luz (FOLEY, 1993). Este obstáculo poderia ser um objeto opaco que impede a passagem de luz.

Para o cálculo de sombras, é utilizado um raio adicional, denominado "raio sombra", que é disparado para unir um ponto de interseção do objeto atingido com um ponto na fonte de luz. Isto é mostrado na Figura 4, retirada de (APPEL, 1968). Se um destes raios sombra atinge um obstáculo durante sua trajetória, então o objeto está na sombra naquele determinado ponto. Se existir mais de uma fonte de luz neste cenário, o processo deve ser calculado da mesma forma para cada uma das fontes de luz.

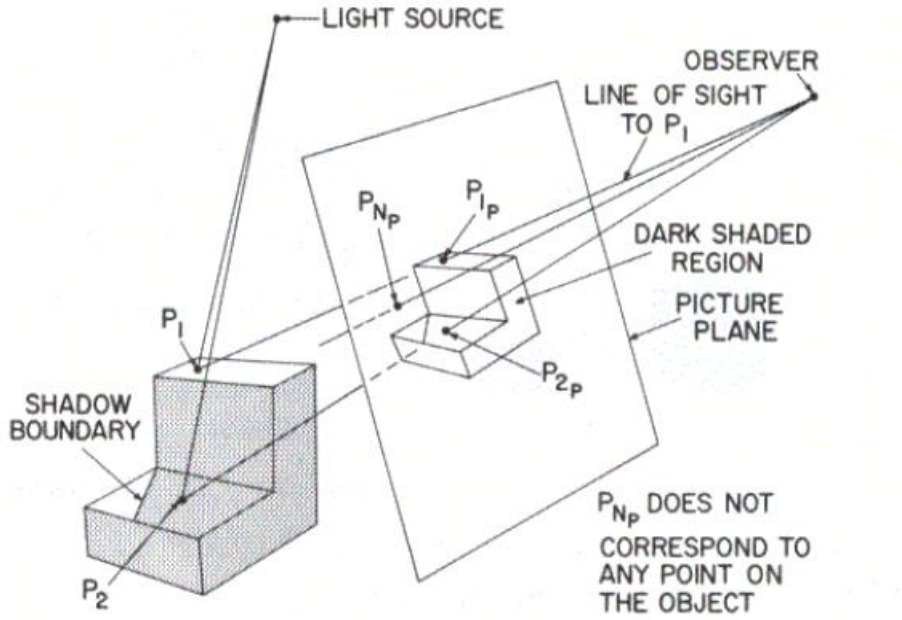

Figura 4: Determinação se um ponto no objeto está na sombra. Fonte: (APPEL, 1968)

3.2.2 Reflexão

Este efeito ocorre quando um raio atinge uma superfície refletora, como um espelho, permitindo que sua trajetória seja desviada em direção à região de onde este raio é oriundo (SILVA, 2011). No algoritmo Ray Tracing, quando o raio visual intersepta um ponto na superfície refletora, cria-se um novo raio que é lançado, a partir deste ponto, na direção contrária da reflexão. Pinta-se, então, este ponto com a cor calculada a partir do raio refletido.

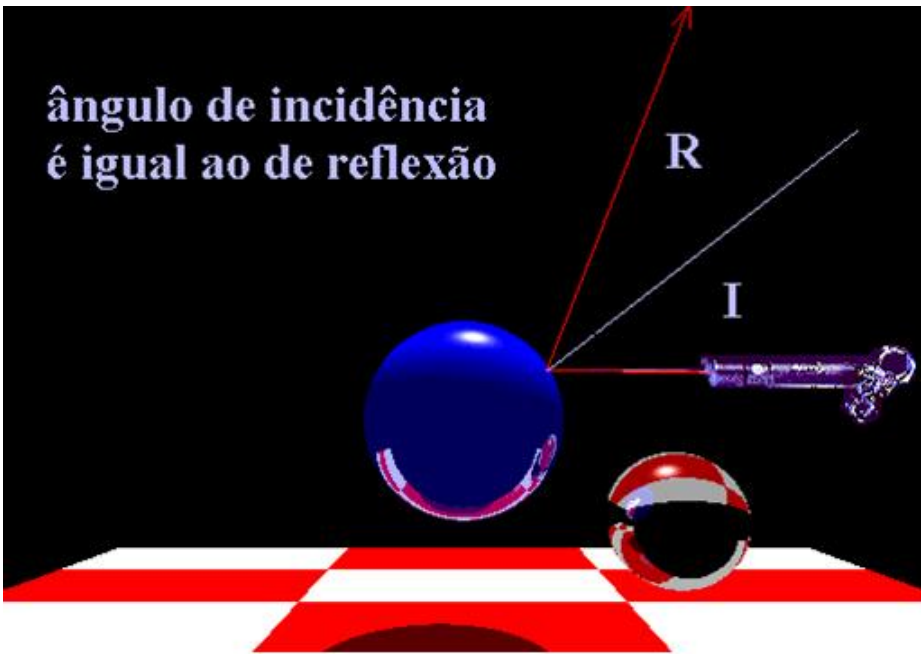

Figura 5: Reflexão de um raio em um objeto. Fonte: (FRANCISCO, 2011)

3.2.3 Transparência

Algumas das superfícies podem ser formadas por materiais translúcidos ou transparentes. Em materiais translúcidos, os raios luminosos atravessam a superfície sofrendo um efeito de reflexão difusa, o que permite que não ocorra uma visão completa do material. Já em materiais transparentes, como o vidro, podem ser vistos outros objetos através deste material, embora, na maioria dos casos, os raios que penetram neste material sejam refratados. O algoritmo Ray Tracing permite determinar a cor do ponto na superfície, no instante em que o raio visual atingir o objeto transparente, a partir da cor do raio auxiliar criado pelo algoritmo que segue a direção da refração (T) (SILVA, 2011). A Equação 3.10 permite calcular o raio refratado T:

$$T = (N_{21}.(N.I) - \sqrt{1 - N_{21}^2.(1 - (N.I)^2).N - N_{21}.I}) \tag{3.10}$$

Onde

$N_{21} = \frac{N_2}{N_1}$

N = Vetor normal que atingiu o ponto na superfície

I = Vetor de incidência da luz

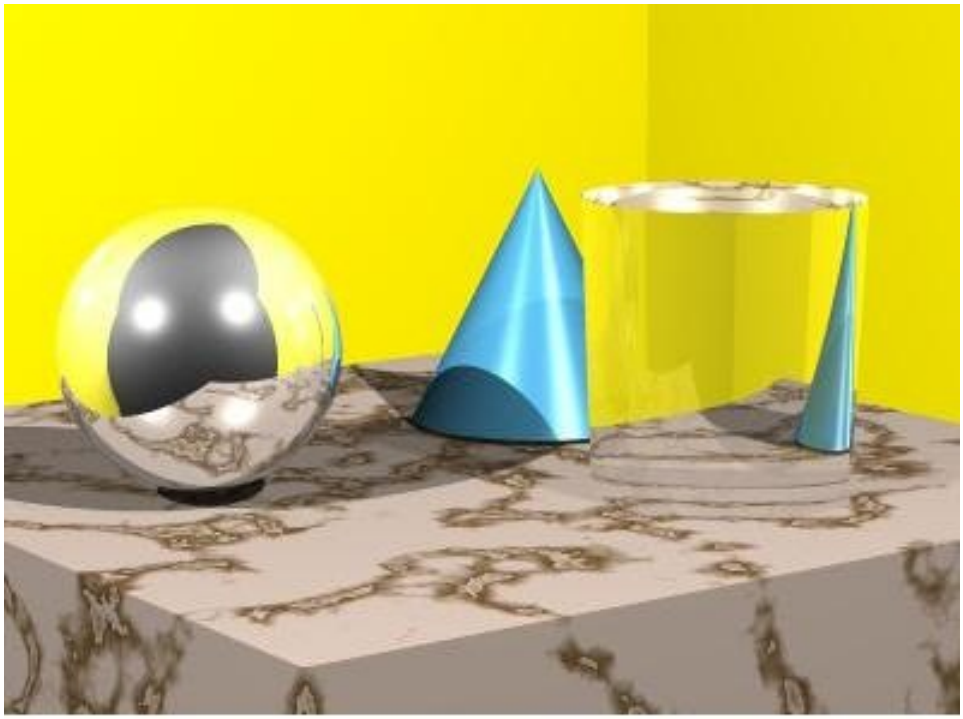

Figura 6: Imagem gerada pelo Ray Tracing com o efeito de transparência. Fonte: (VALCAR.NET, 2011)

A Figura 6 foi gerada pelo Ray Tracing sendo representados 4 objetos em uma cena: uma mesa de mármore, um cilindro de vidro, um cone azul e uma esfera cromada, esta última refletindo todo o ambiente (a mesa, o cone e o cilindro).

3.2.4 Algoritmo recursivo

O algoritmo mostrado abaixo, (VALCAR.NET, 2011), suporta os efeitos ocorridos no Ray Tracing: sombra, reflexão e transparência. Neste algoritmo, os Passos 4 e 5 são recursivos, uma vez que encontrada a interseção entre o raio auxiliar gerado e o objeto, todos os passos (1, 2, 3, 4 e 5) são executados novamente.

Para cada ponto da imagem faça:

1 - Verificar qual objeto se encontra mais próximo do observador a partir deste ponto.

2 - Se existir fontes de luz iluminando este ponto no objeto então:

Para cada fonte de luz faça:

Se existir um objeto entre a fonte de luz e o ponto analisado então:

Pintar o ponto na tela com a cor da intensidade da luz no ambiente.

Senão:

Pintar o ponto na tela com a cor normal do objeto.

3 - Senão:

O objeto está na sombra.

4 - Se a superfície for refletora então:

Criar um raio auxiliar a partir deste ponto, em direção contrária ao reflexo.

Pintar este ponto com a cor calculada no raio auxiliar refletido.

5 - Se a superfície foi transparente então:

Criar um raio auxiliar a partir deste ponto, em direção à refração.

Pintar este ponto com a cor calculada no raio auxiliar refratado.

3.3 Iluminação

Atualmente, a computação gráfica consegue gerar um alto grau de foto-realismo (VALCAR.NET, 2011). Isto se deve ao fato de existirem modelos matemáticos que descrevem a forma como ocorrem as interações dos raios luminosos nas superfícies dos objetos. Nesta seção, serão abordados três modelos de iluminação: Modelo da Reflexão Ambiente, Modelo da Reflexão Difusa e Modelo de Phong para o Brilho Especular.

3.3.1 Modelo da reflexão ambiente

Neste modelo, que busca simular as reflexões mútuas entre os objetos, a iluminação do ambiente torna-se uma fundamental variável na equação matemática, permitindo que os objetos possuam luz própria, e comecem a transmitir luz, mesmo que esta luz emitida não necessariamente ilumine outros objetos (VALCAR.NET, 2011). A intensidade da iluminação ambiente é dada por:

$$I = I_a.K_a \tag{3.11}$$

onde I_a corresponde à intensidade de luz no ambiente, sendo presente em todos os objetos. K_a é o coeficiente de reflexão ambiente que determina a quantidade de iluminação ambiente que é refletida a partir da superfície do objeto, variando de 0 a 1.

3.3.2 Modelo da reflexão difusa

Neste modelo, a reflexão difusa ocorrida em determinados pontos das superfícies dos objetos segue a lei de Lambert, que diz que a intensidade dos raios refletidos, por unidade de área,

não dependem da direção da reflexão. A reflexão difusa é dada por:

$$I = k_a . \sum_n I_n . cos\theta_n \tag{3.12}$$

onde I_n corresponde à intensidade da n-ésima fonte de luz e θ_n é o ângulo de incidência desta luz em relação ao vetor normal à superfície.

3.3.3 Modelo de Phong para o brilho especular

O modelo de Phong simula o efeito causado pelas superfícies que possuem uma textura brilhosa. Estas superfícies refletem os raios emitidos pelas fontes de luz, provocando um espalhamento da luz em torno do reflexo. Este espalhamento depende do aumento da intensidade de luz, próximas das direções de reflexão especular R, conforme mostrado pela Figura 7. Esta intensidade de luz está relacionada com o vetor de posição V do observador, sendo máxima na direção do vetor de reflexão especular R.

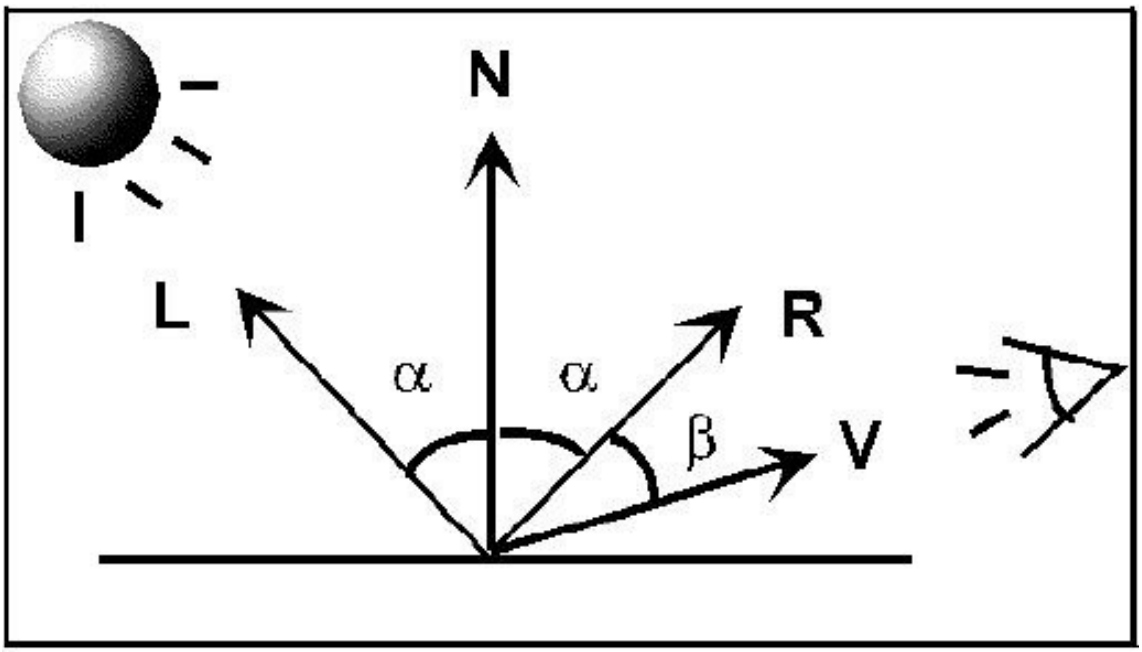

Figura 7: Modelo de iluminação de Phong. Fonte: (SILVA, 2011)

Na Figura 7, L representa o vetor de incidência de luz, N é o vetor normal, R é o vetor de reflexão especular e V é o vetor que indica a posição do observador. A equação de iluminação, segundo o modelo de Phong, é descrita da seguinte forma:

$$I = I_a . K_a + \sum_{i=1}^{i=m} I_{fonte_i} . [K_d . (N . L_i) + K_e . (R_i . V)] \tag{3.13}$$

onde,

I_a é a intensidade de luz no ambiente.
K_a é o coeficiente da iluminação no ambiente.
I_{fonte_i} é a intensidade da i-ésima fonte de luz.
K_d é o coeficiente da reflexão difusa.
K_e é o coeficiente da reflexão especular.

4 ANÁLISE EXPERIMENTAL

Este capítulo descreve a metodologia experimental utilizada e os resultados encontrados com os experimentos realizados com as métricas OO. Inicialmente, será feita uma descrição breve do processador FemtoJava (ITO, 2000), utilizado como referência para a coleta das métricas físicas como quantidade de ciclos de execução e consumo energético. Também serão apresentadas as ferramentas de instrumentação utilizadas neste trabalho, como a ferramenta CAT (NEVES, 2005), utilizada para facilitar a coleta do consumo de memória, e a ferramenta BIT (LEE; ZORN, 2011), para auxiliar na coleta das métricas de consumo energético e número de ciclos. Após isso, serão descritas as etapas realizadas antes do desenvolvimento deste TCC, assim como os resultados encontrados para as métricas de eficiência e para as métricas de qualidade de software para sete versões paralelas desenvolvidas a partir do RayTracer (SMITH; BULL; OBDRZáLEK, 2001). Além disso, serão abordados os experimentos propostos para este Trabalho de Conclusão de Curso (TCC), com o objetivo de apresentar todas as etapas desenvolvidas, juntamente com os resultados obtidos.

4.1 Processador FemtoJava

O FemtoJava (ITO, 2000) é um microprocessador, utilizado em sistema embarcados, que permite realizar o processamento nativo das aplicações Java, uma vez que as instruções executadas por este processador são as próprias instruções do código *assembly* Java (*bytecodes* Java). Deste modo, elimina-se a utilização de máquinas virtuais que tornam a execução mais lenta e requerem espaço de armazenamento, limitando assim a aplicabilidade do Java em alguns domínios de aplicação como o de sistemas embarcados, de sistemas de tempo-real, ou mesmo de aplicações paralelas de alto desempenho.

Algumas características deste processador (NEVES, 2005):

- Suporta apenas uma única *thread*;
- Processa dados exclusivamente do tipo inteiro;
- Não apresenta registradores de propósito geral acessíveis ao programador;
- Arquitetura do processador gerada automaticamente pelo ambiente SASHIMI, permitindo que o desenvolvedor escolha a largura dos dados (caminho de dados e memória) a ser utilizada;

- Apresenta dois tipos de memória internas: a RAM, para a pilha de dados e a ROM para o armazenamento das instruções do programa (*bytecodes*);
- O tamanho destas memórias pode ser configurável;
- Os dados, armazenados na RAM, são alocados em tempo de projeto, ou seja, não possui suporte em nível de *hardware* para alocação de memória em tempo de execução;
- As instruções do programa, que foram compiladas a partir do código-fonte da aplicação Java, são extraídas e alocadas na memória ROM com o auxílio do SASHIMI.

4.2 Ferramentas de instrumentação

Ferramentas de instrumentação permitem alterar uma aplicação, modificando-a, em nível de código fonte ou em nível de código de montagem. Entre as modificações mais comuns produzidas através desta classe de ferramentas, estão a inserção de novas classes, variáveis e novos métodos, de forma automatizada, possibilitando assim uma modificação do estado ou do comportamento da aplicação instrumentada. A Figura 8 mostra algumas das principais ferramentas de instrumentação, juntamente com a ferramenta utilizada no escopo deste trabalho.

Figura 8: Ferramentas de instrumentação de código. Fonte: (NEVES, 2005)

4.2.1 Ferramenta BIT

A ferramenta BIT (LEE; ZORN, 2011) permite modificar o comportamento de uma aplicação de forma a construir uma ferramenta (PAT - *Power Analisys Tool*) de análise energética. A PAT faz referência a uma tabela contendo todos os *bytecodes* executados em um processador embarcado, o FemtoJava. Para cada *bytecode* Java disponível na ISA, existe um valor associado

em termos de consumo energético, medido em *nJ*, e outro em termos de número de ciclos gastos pelo processador FemtoJava para executar o referido *bytecode*. A Tabela 1 apresenta um subconjunto de *bytecodes* juntamente com seu consumo energético e o número de ciclos que o FemtoJava demandaria para os executar.

Tabela 1: Consumo de energia e o número de ciclos de um subconjunto de *bytecodes* Java.

Bytecodes	Consumo de Energia (*nJ*)	Total de Número de Ciclos
iload	4802	1
istore	5970	1
iconst	1999	1
iadd	1848	1
return	31666	9
getstatic	7443	1
putstatic	7453	1
getfield	7453	1
getfield	7453	1
invokevirtual	34003	12
invokestatic	34003	12
new	0	150

A ferramenta BIT trabalha diretamente com os arquivos *.class* da aplicação Java, gerados pelo compilador, inserindo para cada um dos *bytecodes* presentes no *.class* instruções que permitam contabilizar a quantidade de vezes que cada um dos *bytecodes* é executado. Como forma de ilustrar este funcionamento, o Código 1 apresentada um pequeno fragmento de código Java que realiza a soma de dois valores inteiros.

```
1  /*
2  *   Código 1
3  */
4  public class Soma {
5    public static void add () {
6      int a = 4;
7      int b = 4;
8      int r = a + b;
9    }
10 }
```

O Código 2, apresentado a seguir, corresponde a um arquivo *.jad* gerado com o auxílio da ferramenta DJ (DJ, 2011), que será detalhada mais a frente. A ação realizada pela ferramenta BIT sobre o Código 2, mostrado acima, consiste em inserir instruções específicas que permitirão contabilizar o número de vezes que cada um dos *bytecodes*, apresentados no Código 1, foi

executado.

```
/*
*   Código 2
*/
public class Soma {
 public soma () {
 // 0 0:aload_0
 // 1 1:invokespecial   #1 <Method void Object()>
 // 2 4:return
 }
 public static void add () {
  byte byte0 = 4;
 // 0 0:iconst_4
 // 1 1:istore_0
  byte byte1 = 4;
 // 2 2:iconst_4
 // 3 3:istore_1
  int i = byte0 + byte1;
 // 4 4:iload_0
 // 5 5:iload_1
 // 6 6:iadd
 // 7 7:istore_2
 //  8 8:return
 }
}
```

Com base nestes *bytecodes*, a ferramenta PAT analisará a tabela contendo todos os *bytecodes*, semelhante à Tabela 1, buscando selecionar nesta tabela as instruções presentes no *.class* da aplicação. Após isso, multiplica-se o valor gerado por cada uma das instruções de contabilização pelo resultado encontrado no consumo energético dos *bytecodes* selecionados na tabela. Assim, a ferramenta PAT fará o somatório do consumo energético gasto por cada um dos *bytecodes* selecionados. O cálculo para o número de ciclos gastos é feito da mesma forma que o do consumo energético. A Tabela 2 apresenta os dados para o consumo energético, o número de ciclos gastos e o número de *bytecodes* executados, encontrados para a aplicação Soma.

Tabela 2: Métricas de eficiência coletadas para a aplicação Soma.

Aplicação	Ciclos (Total)	Energia (*nJ*)	Instruções (Total)
Soma	17	6,2688285	9

4.2.2 Ferramenta CAT

A ferramenta CAT (NEVES, 2005) permite adaptar o código-fonte de uma aplicação Java analisada a partir dos seus arquivos *.class*, substituindo os *bytecodes* relacionados ao gerenciamento dinâmico de memória, os quais não são suportados pelo processador FemtoJava, por *bytecodes* que são suportados. Estes são disponíveis através de métodos de gerenciamento de memória presentes na DMML (*Dynamic Memory Management Library*). Assim, ela substitui os *bytecodes* não suportados por chamadas de métodos contidos na DMML. A DMML é representada por uma classe, OpBase.java, a qual contém todos os métodos responsáveis pelo gerenciamento de memória, destacando-se métodos responsáveis pela alocação e desalocação de objetos na memória e o acesso à atributos e métodos de objeto, entre outros. A *heap* é representada por um vetor de inteiros com um tamanho fixo nesta classe, sendo incrementada a cada alocação com um valor de uma contabilização do tamanho do objeto. Da mesma forma, a *heap* é decrementada a cada desalocação do valor correspondente ao tamanho do objeto. A referência a cada um desses objetos é armazenada em um vetor separado denominado TDO.

A Figura 9 ilustra uma arquitetura geral da CAT, dividida em duas etapas. A primeira etapa, à esquerda, ocorre sobre o código da aplicação onde é feita a substituição dos *bytecodes* não suportados pelo FemtoJava por chamadas de métodos armazenados na DMML. A segunda etapa é realizada sobre os arquivos *.class* gerando a classe MemDatApp.java, onde estão contidas todas as *constant pools* da aplicação alvo, juntamente com as informações sobre a estrutura de cada um dos objetos. Estas *constant pools* consistem em tabelas que possuem as informações necessárias para a execução de um código Java, sendo que cada classe possui sua própria *constant pool* (NEVES, 2005).

Para o funcionamento desta ferramenta, os arquivos *.class* da aplicação devem ser enviados para a ferramenta DJ (Decompiler Java). Esta ferramenta tem como principal função gerar, a partir de um arquivo *.class*, um arquivo *.jad* contendo o código-fonte referente ao código Java original da aplicação. Este arquivo *.jad* possui, além das instruções da linguagem Java, os *bytecodes* necessários para que cada um dos comandos Java em nível de código fonte possam ser executados conforme mostrado no Código 2. Após isso, este formato de arquivo é enviado para a ferramenta CAT onde passará por uma adaptação de código, com o objetivo de descobrir os *bytecodes* não suportados pelo FemtoJava. Isso levará a uma modificação nas linhas de código Java, referentes ao uso destes *bytecodes*, para incorporar chamadas aos métodos da DMML que tratarão da implementação correspondente a cada um dos *bytecodes*, gerando desta forma, uma aplicação que quando compilada conterá apenas as instruções pertencentes a um conjunto pré-definido de *bytecodes* suportados pelo FemtoJava. A Figura 10 apresenta o fluxo de execução

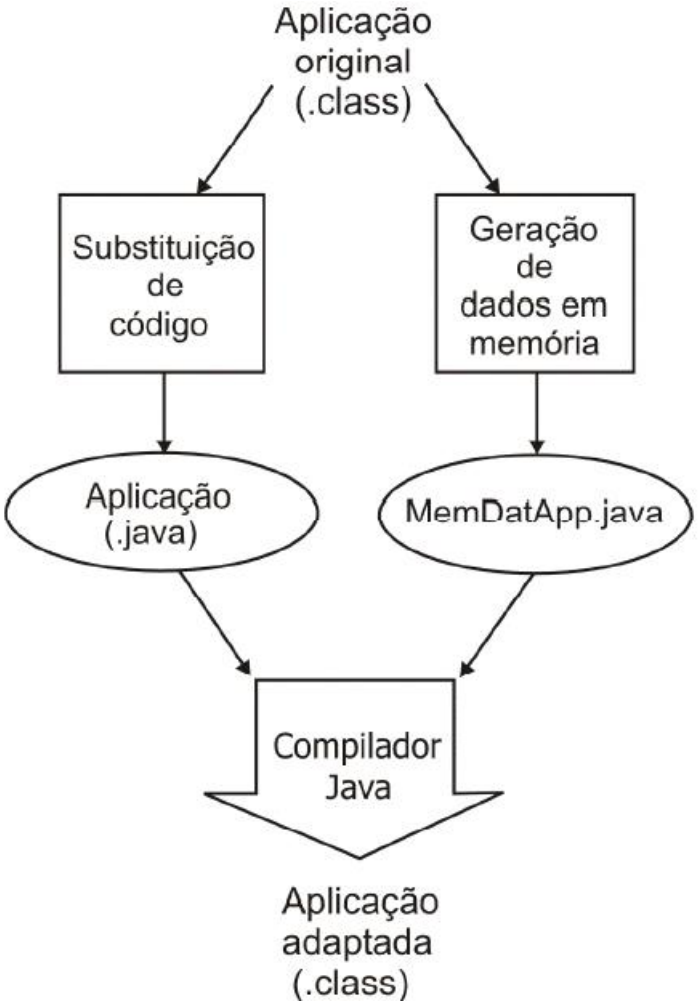

Figura 9: Arquitetura geral da CAT. Fonte: (NEVES, 2005)

da ferramenta CAT.

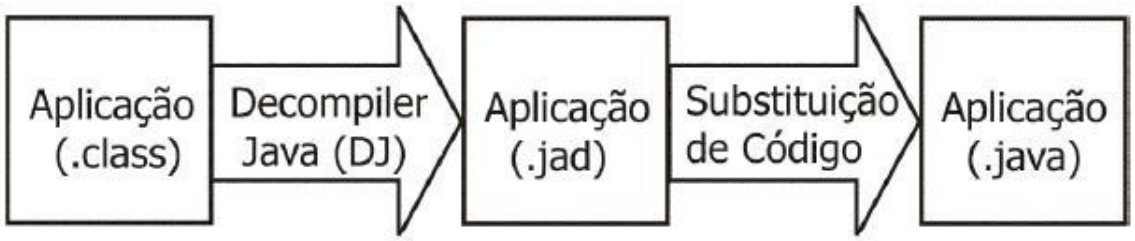

Figura 10: Fluxo de execução da ferramenta CAT. Fonte: (NEVES, 2005)

O Código 3 representa um trecho de um arquivo *.jad* contendo um algoritmo simples. O código subsequente à este, mostrado no Código 4, representa o mesmo trecho de código, em Java, porém contendo instruções adaptadas pela ferramenta CAT.

```
1 /*
2 *   Código 3
```

```
*/
public static void main(String args[]) {
 byte byte0 = 4;
//  0  0:iconst_4
//  1  1:istore_1
 byte byte1 = 2;
//  2  2:iconst_2
//  3  3:istore_2
 Soma soma = new Soma();
//  4  4:new           #2  <Class Soma>
//  5  7:dup
//  6  8:invokespecial  #3  <Method void Soma()>
//  7 11:astore_3
 soma.ADD(byte0, byte1);
//  8 12:aload_3
//  9 13:iload_1
// 10 14:iload_2
// 11 15:invokevirtual  #4  <Method void Soma.ADD(int, int)>
// 12 16:return
}
```

```
/*
*   Código 4
*/
public static void main(String args[]) {
 OpBase.inicializa();
 int soma = 0;
 int byte0 = 4;
 int byte1 = 2;
 soma = OpBase.trataAstore(soma, OpBase.trataNew(2, 0));
 Soma.ADD(4, soma, byte0, byte1);
 OpBase.decCont(soma);
 OpBase.trataReturn(177);
}
```

Na etapa de geração de dados na memória, presente na Figura 9, a classe MemDatApp.java armazena os dados referentes à gerência dinâmica de memória, contendo as *constant pools* de todas as classes da aplicação e as informações vinculadas ao tipo e o posicionamento de cada

uma das variáveis de instância presentes nas classes. Estas informações referentes ao tipo e ao posicionamento são necessárias pois a DMML deve ser capaz de determinar a quantidade de atributos em cada objeto, assim como seus tamanhos, seus corretos posicionamentos (para acesso a estes campos) e se estes campos possuem ou não referências a outros objetos. A forma para acessar a *constant pool* e os campos de uma determinada classe é semelhante ao acesso executado pela JVM (Java Virtual Machine). Entretanto, a única diferença encontrada entre estes dois tipos de acessos é que a CAT utiliza um código numérico que diferencia cada uma das classes da aplicação, o que permite aumentar o desempenho e diminuir o número de comparações de *strings* realizadas pela JVM.

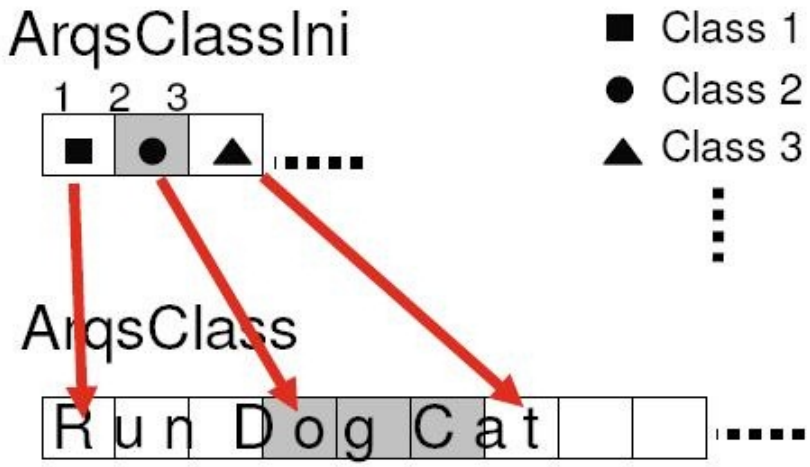

Figura 11: Codificação numérica para a identificação dos objetos. Fonte: (NEVES, 2005)

Na Figura 11, é mostrado como é feito o relacionamento entre cada código numérico e suas respectivas classes. Nesta figura, cada índice do vetor ArqsClassIni armazena um determinado código que indica a correta posição do nome da classe armazenada no vetor ArqsClass. Desta forma, o símbolo quadrado armazenado na primeira posição do vetor ArqsClassIni aponta diretamente para posição onde se inicia o nome da classe *Run* no vetor ArqsClass, o símbolo círculo aponta para a posição onde se inicia o nome da classe *Dog*, e assim por diante. Para identificar o código de uma determinada classe, a DMML deve percorrer cada um dos índices de ArqsClassIni e realizar comparações entre o nome da *String* contendo a classe apontada pelo código correspondente ao índice e a *String* contendo o nome da classe desejada. Após a identificação do código referente a classe buscada, a DMML realiza uma consulta na *constant pool* desta classe através do código numérico encontrado anteriormente que indexará uma das posições do vetor CpsIndexIni, conforme mostrado na Figura 12. O valor contido na posição do vetor CpsIndexIni indexado a partir do código numérico informa o valor de um outro índice (i), usado para acessar uma das posições do vetor CpsIndex. Cada uma das posições do vetor

CpsIndex são usadas para armazenar o endereço de uma das entradas da *constant pool* buscada, que está armazenada no vetor Cps, junto as demais *contant pools* da aplicação. Dessa forma, o último índice calculado (*i*) esta relacionado com o endereço inicial do módulo (*p*) em CpsIndex correspondente com a *constant pool* buscada. Assim, o valor correspondente com a entrada (*e*) da *constant pool*, ao qual deseja-se realizar acesso deve ser usado como um deslocamento a ser somado com o valor calculado correspondente ao endereço inicial (*i*), resultando em um valor que estará relacionado com a posição exata para realizar o acesso no vetor CpsIndex. Por fim, o valor contido nesta posição de CpsIndex indicará à posição (*o*) em Cps onde inicia-se a entrada buscada da *constant pool* alvo.

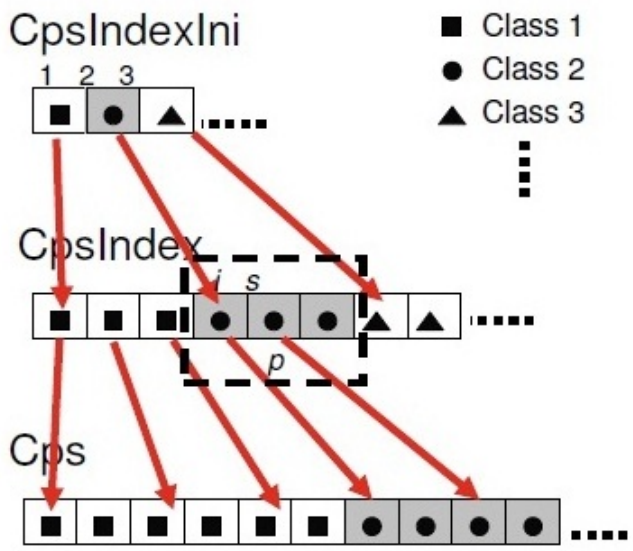

Figura 12: Acesso a *constat pool* de uma classe qualquer. Fonte: (NEVES, 2005)

O acesso aos campos de cada classe é semelhante a forma com que a CAT acessa às *constant pools*, diferenciando-se entretanto por possuir um nível a menos em sua hierarquia. Para realizar o acesso à estrutura de campos de uma determinada classe deve-se primeiramente saber o índice calculado a partir do código da classe. Este código informa a posição do vetor FieldsIni relacionado com a classe a qual o campo desejado pertence. Esta posição em FieldsIni corresponde ao índice relacionado com o endereço inicial da parcela (*p*) em Fields ao qual deseja-se acessar, contendo a classe pretendida. A partir deste momento, o acesso a qualquer campo da classe buscada é feita através de deslocamentos cujos tamanhos são múltiplos do tamanho usado para armazenar cada informação sobre os campos. A Figura 13 apresenta a hierarquia para acessar os campos de uma determinada classe.

Uma vez que a CAT produz adaptações nos arquivos de formato Java, é necessário considerar algumas regras durante o desenvolvimento da aplicação. Dentre as principais regras, pode-se destacar:

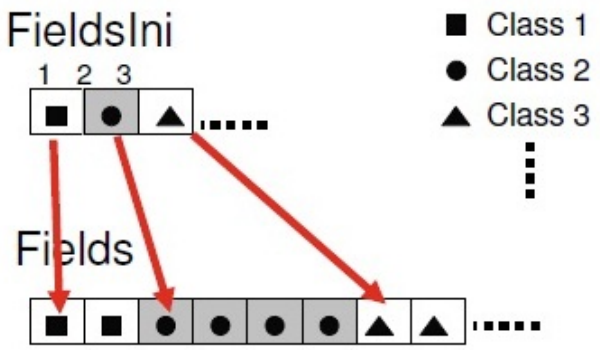

Figura 13: Acesso à estrutura de campos de uma classe qualquer. Fonte: (NEVES, 2005)

- os construtores de cada classe devem ser implementados como métodos de instância comum, onde as variáveis de instância sejam inicializadas apropriadamente;
- a ferramenta não distingue modificadores como *private*, *public*, *protect*, entre outros. Os únicos modificadores reconhecidos são *byte*, *int*, *char*, *arrays* (uni, bi e tri-dimensional), *short* e *string*;
- a ferramenta não possui suporte a alguns elementos de linguagem, tais como herança, polimorfismo, *Interface*, *Exception*, entre outros;
- os conceitos de *package* e *import* não são suportados. Desta forma, todas as classe da aplicação devem estar dispostas no mesmo diretório;
- não é permitido passar variáveis globais como parâmetros de *returns*;
- não é permitido usar métodos em *ifs*;
- não é permitido inicializar as variáveis globais em sua própria declaração.

4.3 Trabalhos anteriores

Esta seção apresentará as etapas que foram desenvolvidas em atividades de iniciação científica anteriores a este TCC. Descreve as metodologias utilizadas e os resultados obtidos para a coleta das métricas de software e das métricas de eficiência.

4.3.1 Etapas iniciais

A primeira etapa desta pesquisa iniciou-se com um estudo aprofundado das métricas de qualidade de software, onde foi selecionado um conjunto de métricas para o desenvolvimento

deste trabalho. Paralelamente a esta etapa, foi realizada uma busca por uma ferramenta que tivesse a capacidade de gerar automaticamente os resultados das métricas de software a partir de uma aplicação alvo. As métricas escolhidas para este estudo, já explicadas no Capítulo 2, foram: Abstração (A), Profundidade de Herança (DIT), Falta de Coesão de Métodos (LCOM), Acoplamento Aferente (Ca), Acoplamento Eferente (Ce), Número de Parâmetros (NOP), Número de Métodos (NOM), Número de Métodos Estáticos (NOSM) e Número de Atributos Estáticos (NOSA). Estas métricas foram selecionadas por estarem voltadas para aplicações orientadas a objetos e por estarem relacionadas com as métricas de qualidade, como reuso de software e manutenção. Por exemplo, as métricas de Abstração e Profundidade de Herança estão relacionadas com a capacidade de reuso de um software. Para a geração automatizada dos dados referentes a estas métricas, foi escolhido o *plugin* Metrics (METRICS, 2011).

Na etapa seguinte, realizou-se uma análise de várias aplicações embarcadas que tivessem a capacidade de paralelização implementadas, onde uma foi selecionada para servir de base para a realização do estudo aprofundado sobre o co-relacionamento de métricas de qualidade de software e métricas de eficiência. A aplicação escolhida foi o RayTracer (SMITH; BULL; OBDRZáLEK, 2001), descrito no Capítulo 3, por ser uma aplicação embarcada paralela. Este algoritmo escolhido é desenvolvido na linguagem de programação Java.

Posteriormente, foi desenvolvida uma série de modificações diretamente no código-fonte do RayTracer, sem a alteração do grau de paralelismo. Estas modificações permitiram gerar sete diferentes versões do algoritmo, onde cada uma dessas versões busca, de forma objetiva, dar ênfase exclusiva é variação de cada uma das métricas de software estudadas neste trabalho. Estas versões são mostradas na Tabela 3.

Tabela 3: Versões do RayTracer e suas respectivas descrições.

Versão do RayTracer	Descrição
RTOriginal	Versão original do RayTracer
RTA	Redução de abstração
RTCeCa	Aumento do Ca e redução do Ce
RTDIT	Redução na profundidade de herança
RTLCOM	Aumento da coesão nos métodos
RTNOM	Redução do número de métodos
RTNOP	Redução do número de parâmetros
RTNOSM	Aumento do número de métodos estáticos

Cada uma destas versões foi obtida com a variação exclusiva das métricas de software selecionadas, com exceção da métrica NOSA que foi utilizada como proposta de implementação para o TCC. Estas variações nas métricas OO foram feitas visando um melhor desempenho nas versões desenvolvidas, ou seja, buscou-se aumentar ou diminuir ao máximo cada uma das

métricas escolhidas. Por exemplo, diminuiu-se o valor da métrica de Abstração com o objetivo de melhorar o desempenho da versão relacionada com esta métrica (RTA). Abaixo, segue uma descrição breve das modificações necessárias para a geração destas sete versões.

Versão RTA: para a geração da versão RTA, foi necessário diminuir o nível de abstração da versão original do RayTracer. Para isso, teve-se que transferir todos os atributos e métodos das classes abstratas, da versão original, para as classes que às estendem. Além disso, os métodos abstratos, das classes abstratas, acabaram sendo implementados também nas classes que às herdam. As *Interfaces* também sofreram algo semelhante. Implementou-se todos os atributos e métodos, presentes nas *Interfaces*, nas classes que às implementam. Dessa forma, foi possível eliminar todas as classes abstratas e as *Interfaces*, da versão original, na versão RTA, reduzindo assim o nível de abstração.

Versão RTDIT: para a geração da versão RTDIT, onde houve uma redução no número de heranças, foi necessário implementar todos os atributos e métodos das classes herdadas nas classes que herdam. Também foi necessário transferir todas as referências existentes para as classes herdadas para as classes que às estendem. Assim, excluiu-se todas as classes herdadas o que permitiu uma redução na profundidade de herança.

Versão RTLCOM: para a geração da versão RTLCOM, onde houve um aumento na falta de coesão entre os métodos, teve-se que agrupar todas as classes relacionadas em uma única classe. Para reduzir a falta de coesão, o processo é feito ao contrário, ou seja, divide-se a classe em várias subclasses para aumentar a coesão nos métodos.

Versão RTCeCa: para a geração desta versão, onde houve um aumento no acoplamento aferente e uma redução no acoplamento eferente em relação a versão original (RTOriginal), foi necessário atribuir cada classe à um pacote específico.

Versão RTNOP: para a geração desta versão, onde houve uma redução no número de parâmetros, foi preciso aumentar o número de variáveis globais (atributos) para que ocorresse esta redução.

Versão RTNOM: para a geração da versão RTNOM, onde buscou-se diminuir o número de métodos, foi necessário agrupar vários métodos em um único método. Para isso, teve-se que substituir a maioria das chamadas de métodos pelo corpo dos métodos que estavam sendo chamados, eliminando-se assim, estes métodos. Isso também permitiu que houvesse um aumento no número de linhas de código por método.

Versão RTNOSM: para a geração da versão RTNOSM, onde houve um aumento no número de métodos estáticos, teve-se que adicionar, na maioria dos métodos, o modificador *static*. Em

seguida, foi necessário substituir todos os acessos à esses métodos, uma vez que agora eles são acessados a partir da classe, e não mais do objeto instanciado.

4.3.2 Coleta das métricas de software

Para a coleta das métricas de software selecionadas, foi utilizada a plataforma Eclipse, que é um ambiente usado para a programação de softwares, onde foi instalado o *plugin* Metrics. Este *plugin* permite gerar os valores para cada uma das métricas orientadas a objetos a partir de uma aplicação alvo. Dessa forma, as métricas de software escolhidas, com exceção da métrica NOSA por não ter sido desenvolvida até esta etapa, foram coletas para cada uma das versões desenvolvidas do RayTracer. Estes valores estão presentes na Tabela 4.

Tabela 4: Métricas de qualidade de sofware para cada uma das versões do RayTracer.

Versões	A	Ca	Ce	DIT	LCOM	NOM	NOP	NOSM
RTOriginal	0,083	1	1	1,21	0,207	76	1,294	9
RTA	0	1	1	1	0,299	78	1,449	8
RTCeCa	0,167	2,611	0,889	1,21	0,207	76	1,355	9
RTDIT	0,033	1	1	1	0,029	78	1,443	8
RTLCOM	0,125	1	1	1	0,471	72	1,611	9
RTNOM	0,125	1	1	1	0,581	40	1,795	4
RTNOP	0,083	1	1	1,21	0,257	76	0,91	9
RTNOSM	0,083	1	1	1,21	0,075	46	1,635	39

Nesta tabela, a métrica Abstração (A) é representada pelo valor médio, medida através do número de classes abstratas e interfaces dividido pelo número total de classes concretas em um pacote. A métrica Falta de Coesão nos Métodos (LCOM), é calculada com a utilização do método Henderson-Sellers. Neste método, se *m(Aj)* representa o número de métodos acessando cada atributo *A*, calcula-se a média de todos os atributos em *m(Aj)*, subtrai-se o número de métodos *m* e divide-se este resultado por *(1-m)*. A métrica de Profundidade de Herança (DIT) corresponde à distância máxima da classe até a raiz da árvore. A métrica Número de Parâmetros (NOP) é calculada através da média dividido pelo valor máximo por método. Essa média é calculada com a soma de todos os parâmetros de um método, fazendo-se assim, a média do número de parâmetros de todos os métodos de uma classe. A métrica Acoplamento Aferente (Ca) é calculada através da média do número de classes externas que se comunicam com as classes internas a um determinado pacote. A métrica Acoplamento Eferente (Ce) é calculada através da média do número de classes internas que se comunicam com as classes externas a um determinado pacote. As métricas Número de Métodos (NOM) e Número de Métodos Estáticos (NOSM) representam o valor ao todo na aplicação.

Esta coleta permitiu que houvesse uma identificação no relacionamento entre as métricas orientadas a objetos, onde a alteração de uma produziu uma alteração em um subconjunto de outras métricas. A exemplo disto, percebeu-se que diminuindo a profundidade de herança, muitas vezes diminui-se também o nível de abstração. Neste exemplo é mostrado que quando compara-se as versões RTOriginal e RTDIT, onde a profundidade de herança reduziu respectivamente de 1,211 para 1, reduz-se também o nível de abstração, de 0,083 para 0,033, respectivamente. Isto é possível uma vez que as classes herdadas que foram eliminadas podem ser abstratas. Uma outra métrica que é afetada quando ocorre uma redução no número de heranças é o número de linhas no código, não analisada neste trabalho. Identificou-se que reduzindo a profundidade de herança, pode ocorrer um aumento no número de linha de código. As mesmas versões citadas anteriormente (RTOriginal e RTDIT), quando comparadas, apresentaram os seguintes valores: a diminuição de número de heranças respectivamente de 1,211 para 1 gerou um aumento no número de linhas de código, de 743 para 932, respectivamente.

4.3.3 Coleta das métricas de eficiência

Para a coleta das métricas de eficiência, foi necessário utilizar a ferramenta PAT, o que possibilitou coletar as métricas de eficiência para cada uma das versões do RayTracer, que são apresentados na Tabela 5. Pode-se concluir, a partir desta tabela, que a versão RTNOM foi a que obteve o melhor desempenho dentre todas as versões, uma vez que o consumo de energia foi menor, provocado pelo menor número de *bytecodes* executados e o menor número de ciclos gastos pela versão. A versão RTNOSA novamente não aparece nesta tabela por não ter sido implementada até esta etapa.

Tabela 5: Métricas de eficiência para cada uma das versões do RayTracer.

Versões	Ciclos (Total)	Energia (*nJ*)	Instruções (Total)
RTOriginal	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}
RTA	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}
RTCeCa	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}
RTDIT	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}
RTLCOM	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}
RTNOM	1,34 x 10^{11}	0,73 x 10^{11}	1,01 x 10^{11}
RTNOP	2,49 x 10^{11}	1,17 x 10^{11}	1,41 x 10^{11}
RTNOSM	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}

4.4 Trabalho proposto

Esta seção apresentará as etapas propostas para o desenvolvimento deste trabalho. Inicialmente, será introduzida a metodologia utilizada, apresentando a seguir uma descrição de cada uma das etapas concluídas neste trabalho.

4.4.1 Metodologia

Abaixo, serão apresentadas as etapas propostas para o desenvolvimento deste TCC.

- Inicialmente, foi realizado um estudo detalhado da aplicação original. Após isso, desenvolveu-se duas versões do RayTracer, utilizando a linguagem de programação Java, onde ambas possuem ênfase na métrica de qualidade relacionada com o número de atributos estáticos (NOSA), com o objetivo de alterar os atributos em todas as classes da versão original, transformando-os em estáticos. Desse modo, o principal objetivo é avaliar os impactos causados pela variação da métrica NOSA sobre as métricas de eficiência analisadas neste TCC, buscando comparar os resultados encontrados para esta versão estática com os resultados encontrados para as versões dinâmicas do RayTracer, desenvolvidas anteriormente. A principal diferença entre uma versão dinâmica e uma versão estática é que na dinâmica as classes instanciam os objetos durante a execução da aplicação, enquanto que na estática não existe a criação de objetos, ou seja, as variáveis são alocadas na memória juntamente com as classes durante a execução da aplicação. A primeira versão desenvolvida, chamada RTNOSA_T, não possuía suporte a *multithreading*, uma vez que o foco desta implementação era justamente converter todos os atributos, da aplicação original, para estáticos e verificar se os resultados obtidos estavam corretos, para uma única *thread* (*thread* do método *main*). Na segunda versão desenvolvida, acrescentou-se paralelismo, com suporte a mais de uma *thread*, sendo chamada de RTNOSA_MT.

- Como um passo seguinte, foram extraídas as métricas de qualidade de software e as métricas de eficiência das duas versões NOSA desenvolvidas no passo anterior. As métricas de qualidade foram coletas com a utilização do *plugin* Metrics, já citado anteriormente. As métricas físicas de eficiência, como o número total de ciclos de execução, o consumo de energia em *nJ* e o número total de instruções executadas, foram extraídos com a utilização de uma ferramenta de instrumentação, também já mencionada. Isso foi feito para que fosse possível observar as influências do uso de atributos estáticos sobre a eficiência da aplicação alvo.

- Na etapa seguinte, foi realizada a extração do consumo de memória de todas as versões desenvolvidas. Para isso, fez-se um estudo de duas ferramentas, o Yourkit Java Profiler (YJP, 2011) e o JProfiler (JP, 2011), onde verificou-se que os resultados gerados por estes softwares não correspondem aos resultados esperados, conforme será abordado mais a frente. Desta forma, optou-se em utilizar a ferramenta CAT, explicada anteriormente, para auxiliar na coleta dos dados de memória.

- Na última etapa, foi feita uma análise buscando encontrar justificativas para os impactos causados nas mudanças das métricas de qualidade sobre o desempenho da aplicação. Para isso, fez-se uma análise da relação entre cada uma das métricas de qualidade geradas com a quantidade de ciclos de energia e de memória consumidos por cada versão, visando:

  – conhecer a fundo as relações existentes entre as variações produzidas em alto nível e as repercussões sobre o desempenho essencialmente inerente a execução dos *bytecodes* sobre a máquina Java.

  – determinar qual ou quais métricas de qualidade causam maior interferência sobre o desempenho da aplicação, assim como a origem destas influências.

4.4.2 Desenvolvimento da versão RTNOSA com uma única *thread*

O desenvolvimento desta versão começou a partir de uma análise da versão original do RayTracer. Como o objetivo era converter para estático o máximo possível dos atributos presentes na aplicação, o que transformaria a versão dinâmica do RayTracer original em estática, foi necessário contabilizar o número de vezes em que cada classe instanciava seus objetos. Para isso, foi inserido, logo abaixo de cada comando *new* do código fonte Java (instrução esta que permite criar objetos relacionados a uma classe específica), instruções que incrementavam cada vez que um objeto era instanciado. Desta forma, obteve-se o número de vezes em que cada uma das classes instanciou seus objetos.

A análise dos dados mostrou que algumas classes criavam seus objetos uma única vez, o que permitiu que seus atributos fossem modificados para estáticos sem causar alterações na execução da versão. Após isso, tratou-se de alterar as classes que possuíam mais de um objeto instanciado. Os atributos de cada uma destas classes que instanciavam objetos mais de uma vez foram implementados através de vetores, possuindo um tamanho igual ao número de vezes que cada uma das classes tinham seus objetos instanciados por ela mesma ou por outra classe. Em seguida, começou-se a modificar todos os acessos à estas classes alteradas. Nesta primeira versão não houve suporte ao paralelismo, ou seja, apenas uma única *thread* é executada, neste

caso, a *thread* do método *main*. Esta versão com uma única *thread* foi desenvolvida com o objetivo principal de simplificar a implementação e verificar se o resultado final obtido por esta versão, neste caso o *checksum* (variável do RayTracer que calcula a quantidade de objetos renderizados na tela), era o mesmo resultado encontrado pela versão original do RayTracer (RTOriginal). A Tabela 6 apresenta os valores encontrados, por meio do *plugin* Metrics, para as métricas estudadas para cada uma das versões desenvolvidas do RayTracer, incluindo a versão RTNOSA com uma *thread* (RTNOSA_T). Os dados para as demais versões foram novamente apresentados para possibilitar uma comparação visual mais fácil dos resultados obtidos.

Tabela 6: Métricas de qualidade de sofware com a versão RTNOSA_T.

Versões	A	Ca	Ce	DIT	LCOM	NOM	NOP	NOSM	NOSA
RTOriginal	0,083	1	1	1,21	0,207	76	1,294	9	7
RTA	0	1	1	1	0,299	78	1,449	8	9
RTCeCa	0,167	2,611	0,889	1,21	0,207	76	1,355	9	7
RTDIT	0,033	1	1	1	0,029	78	1,443	8	9
RTLCOM	0,125	1	1	1	0,471	72	1,611	9	7
RTNOM	0,125	1	1	1	0,581	40	1,795	4	7
RTNOP	0,083	1	1	1,21	0,257	76	0,91	9	7
RTNOSM	0,083	1	1	1,21	0,075	46	1,635	39	7
RTNOSA_T	0,038	1	1	1,143	0	2	2,36	48	63

Nesta tabela, é possível perceber que houve um aumento significativo no número de atributos estáticos, uma vez que o enfoque em desenvolver esta versão estava justamente no aumento destes tipos de atributos. Entretanto, a variação desta métrica provocou algumas alterações em outras métricas analisadas. Primeiramente, houve uma redução no nível de abstração, causado pela eliminação de algumas classes abstratas com o intuito de simplificar o desenvolvimento da aplicação. Nesta versão, as classes abstratas Barrier e Primitive foram completamente removidas, necessitando que os atributos e os métodos encontrados nestas duas classes fossem implementados nas classes que às herdavam, neste caso, TournamentBarrier e Spheres, respectivamente. A interface Serializable foi mantida nesta versão, o que mostrou que esta versão do RTNOSA com uma *thread* ainda possui um certo grau de abstração. Juntamente com a redução da abstração, houve também uma redução na profundidade de herança, causado pela eliminação das classes abstratas, uma vez que as classes que às estendiam não precisavam mais herdar os atributos e métodos presentes nas classes abstratas, pois eles já estavam implementados nas classes que herdavam. Outras duas métricas que foram afetadas pela variação da NOSA foram as métricas NOM e NOSM, que reduziram e aumentaram, respectivamente. Para que um método possa acessar os atributos estáticos de sua classe, este método precisa ser estático. Como todas as classes desta versão possuíam atributos estáticos, com exceção da classe RayTracerRunner, todos os métodos que acessavam diretamente estes atributos também necessitavam ser

estáticos. Isto possibilitou gerar um aumento na métrica NOSM. Já na métrica NOM, os únicos métodos que não eram estáticos estavam presentes na classe RayTracerRunner, que nesta versão era a única classe que possuía atributos não estáticos. Para finalizar, também houve um aumento na métrica NOP. Isso provavelmente deve-se ao fato de enviar como parâmetros dos métodos, variáveis que eram utilizadas como indexadores dos atributos implementados com vetores. Este motivo talvez explique o aumento encontrado na média do número de parâmetros desta versão.

A Tabela 7 apresenta os valores encontrados, com a utilização da ferramenta BIT, para o consumo de energia, o número total de ciclos e o número de *bytecodes* executados, para cada uma das versões do RayTracer, incluindo a versão RTNOSA com uma *thread* (RTNOSA_T).

Tabela 7: Métricas de eficiência com a versão RTNOSA_T.

Versões	Ciclos (Total)	Energia (*nJ*)	Instruções (Total)
RTOriginal	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}
RTA	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}
RTCeCa	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}
RTDIT	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}
RTLCOM	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}
RTNOM	1,34 x 10^{11}	0,73 x 10^{11}	1,01 x 10^{11}
RTNOP	2,49 x 10^{11}	1,17 x 10^{11}	1,41 x 10^{11}
RTNOSM	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}
RTNOSA_T	3,14 x 10^{11}	1,44 x 10^{11}	1,58 x 10^{11}

Um dos problemas encontrados nesta versão foi que ela levou mais tempo para executar do que a versão RTOriginal. O que esperava-se era que esta versão fosse mais rápida, em termos de execução, quando comparada com as outras versões, o que não ocorreu. A explicação encontrada para esta demora na execução se dá provavelmente ao fato do *overhead* causado pelos acessos aos índices dos vetores dos atributos, ou seja, talvez estes acessos estejam fazendo a diferença no tempo de execução. Esta demora na execução explica os valores altos encontrados para cada uma das métricas da versão RTNOSA_T na Tabela 7. Para esta versão, o consumo de energia, o número de ciclos e o número de instruções foram altos quando comparados com as outras versões.

4.4.3 Desenvolvimento da versão RTNOSA com *multithreading*

Para a geração de uma versão paralelizada do RTNOSA, optou-se em alterar a implementação dos atributos presentes nas classes que possuíam mais de um objeto instanciado. Os vetores destes atributos foram alterados para matrizes como estratégia de tentar simular uma paralelização, uma vez que não encontrou-se outra alternativa, em termos de sincronização entre objetos estáticos, para que esta paralelização ocorresse. Desta forma, cada uma das colunas presentes

nestas matrizes representaria uma *thread*. As linhas destas matrizes representam o tamanho dos vetores, ou seja, o número de objetos instanciados por uma classe. Assim, cada coluna possui seus próprios conjunto de dados levando a uma simulação de cada *thread* executando sobre seus próprios conjuntos de objetos.

A Tabela 8 apresenta os valores obtidos, através do *plugin* Metrics, para as métricas orientadas a objetos estudadas neste trabalho. Nesta tabela, inclui-se a versão RTNOSA *MultiThreading* (RTNOSA_MT).

Tabela 8: Métricas de qualidade de sofware com a versão RTNOSA_MT.

Versões	A	Ca	Ce	DIT	LCOM	NOM	NOP	NOSM	NOSA
RTOriginal	0,083	1	1	1,21	0,207	76	1,294	9	7
RTA	0	1	1	1	0,299	78	1,449	8	9
RTCeCa	0,167	2,611	0,889	1,21	0,207	76	1,355	9	7
RTDIT	0,033	1	1	1	0,029	78	1,443	8	7
RTLCOM	0,125	1	1	1	0,471	72	1,611	9	7
RTNOM	0,125	1	1	1	0,581	40	1,795	4	7
RTNOP	0,083	1	1	1,21	0,257	76	0,91	9	7
RTNOSM	0,083	1	1	1,21	0,075	46	1,635	39	7
RTNOSA_T	0,038	1	1	1,143	0	2	2,36	48	63
RTNOSA_MT	0,038	1	1	1,143	0	9	2,824	42	111

Comparando-se os valores obtidos nesta tabela entre as versões RTNOSA com uma *thread* e RTNOSA *multithreading*, observa-se que as métricas A, Ca, Ce, DIT e LCOM mantem-se iguais. Entretanto, as diferenças entre estas duas versões começam a ocorrer na métrica NOM, onde houve um aumento de 2 para 9 nos números de métodos, respectivamente. Este aumento deve-se ao fato de alguns métodos presentes na versão RTNOSA *multithreading* terem sido acrescidos pelo modificador *synchronized*. Este modificador permite que apenas uma *thread* por vez tenha completo acesso a este método, o que garante uma integridade de valores das diversas variáveis envolvidas neste método. Desta forma, alguns métodos da versão RTNOSA_T foram modificados de *static* para *synchronized* na versão RTNOSA_MT, levando esta última versão a ter um maior valor na métrica NOM em relação ao RTNOSA_T. O aumento ocorrido no número de métodos de 2 para 9 produziu uma redução no número de métodos estáticos de 48 para 42, quando comparadas as versões RTNOSA com uma *thread* e o RTNOSA *multithreading*, respectivamente. Outra métrica diferente, entre estas duas versões, foi a NOP, que teve um leve aumento na média dos números de parâmetros entre métodos. Este aumento, observado para a versão RTNOSA *multithreading*, dá-se agora pelo envio de dois indexadores para os parâmetros dos métodos: um para indexar as linhas das matrizes e outro para indexar as colunas. Para finalizar, houve um grande aumento no número de atributos estáticos, entre estas duas versões. Este aumento, na versão *multithreading*, é explicado pela transformação de algumas variáveis

dos métodos da classe RayTracer em atributos estáticos, demonstrando que esta classe da versão RTNOSA_MT terá mais campos estáticos do que a classe RayTracer da versão com uma *thread*. Estes novos campos estáticos foram implementadas através de vetores com o objetivo de evitar possíveis conflitos de dados.

A Tabela 9 apresenta os dados referentes as métricas de eficiência, coletadas a partir da ferramenta BIT, para cada uma das versões do RayTracer, incluindo a versão RTNOSA *multithreading* (RTNOSA_MT). Detectou-se que o tempo de execução, da versão *multithreading*, praticamente dobrou em relação a versão com uma *thread*. A explicação para isso dá-se pelo fato desta última versão desenvolvida possuir matrizes em vez de vetores, o que possibilitaria uma diferença maior no tempo de execução devido aos constantes acessos às posições destas matrizes. Este aumento no tempo de execução da versão *multithreading* poderia ser o principal motivo dos altos valores encontrados para cada uma das métricas de eficiência, na Tabela 9, dentre todas as versões do RayTracer desenvolvidas.

Tabela 9: Métricas de eficiência com a versão RTNOSA_MT.

Versões	Ciclos (Total)	Energia (*nJ*)	Instruções (Total)
RTOriginal	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}
RTA	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}
RTCeCa	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}
RTDIT	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}
RTLCOM	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}
RTNOM	1,34 x 10^{11}	0,73 x 10^{11}	1,01 x 10^{11}
RTNOP	2,49 x 10^{11}	1,17 x 10^{11}	1,41 x 10^{11}
RTNOSM	2,40 x 10^{11}	1,12 x 10^{11}	1,31 x 10^{11}
RTNOSA_T	3,14 x 10^{11}	1,44 x 10^{11}	1,58 x 10^{11}
RTNOSA_MT	4,68 x 10^{11}	1,97 x 10^{11}	2,03 x 10^{11}

4.4.4 Coleta dos dados de memória

Após ser realizada a coleta das métricas de software e a coleta das métricas de eficiência para as versões desenvolvidas do RTNOSA, a próxima etapa deste trabalho foi realizar a coleta do consumo de memória para todas as versões implementadas do RayTracer. É importante salientar que estes dados de memória devem estar relacionados apenas com a aplicação analisada, não levando em consideração outras bibliotecas importadas por ela. O motivo dessas bibliotecas não estarem sendo analisadas é justamente garantir que sejam analisados apenas os objetos que estão diretamente relacionados com a aplicação. Caso essas bibliotecas fossem analisadas, poderia ocorrer a situação em que 90% do pico máximo de memória alocada corresponda à essas bibliotecas deixando o restante apenas para os objetos próprios da aplicação. Para isso,

buscou-se encontrar uma ferramenta que permitisse identificar o número de objetos alocados ou o máximo de *heap* alocada a cada instante de tempo. Desta forma, foram analisadas as ferramentas Yourkit Java Profiler (YJP, 2011) e JProfiler (JP, 2011). Ambas ferramentas produzem dados como:

- consumo de CPU através de gráficos que mostram a porcentagem de uso do processador.
- gráficos de execução das *threads* que permitem a visualização de quais *threads* estão sendo executadas no momento.
- consumo de memória através de gráficos que mostram objetos sendo alocados e desalocados da *heap*.

Entretanto, os dados de memória gerados por ambas as ferramentas correspondem ao consumo de memória da aplicação alvo juntamente com outras bibliotecas importadas por ela, indo contra ao propósito deste trabalho.

Neste contexto, optou-se por utilizar a ferramenta de instrumentação CAT, citada anteriormente, como forma de auxiliar no processo de coleta do consumo de memória de todas as versões desenvolvidas do RayTracer. Entretanto, para que a ferramenta CAT possa adaptar os arquivos Java passados à ela, seria necessário que estes arquivos estivessem em um formato de codificação de acordo com a ferramenta. Desta forma, buscou-se adaptar todas as versões desenvolvidas do RayTracer (incluindo a versão RTNOSA *multithread*) para um formato em que a CAT pudesse reconhecer. Dentre estas adaptações, podem-se destacar a eliminação de construtores de cada uma das versões, subtituíndo-os por métodos que realizassem as mesmas operações dos construtores. Desta forma, a cada nova instância ocorrida no código, inseria-se uma chamada de método que possuía a mesma função do construtor em questão. Também buscou-se eliminar as ocasiões em que os métodos retornavam variáveis de classe, resolvendo este problema com a criação de variáveis dentro do corpo dos métodos que recebiam o conteúdo destas variáveis de classe, para que assim pudessem ser retornados. Outra alteração feita em cada versão foi a eliminação de classes abstratas e interfaces, fazendo com que todos os atributos e métodos destas classes fossem implementados nas classes que às estendiam ou implementavam. Essa alteração permitiu com que as versões RTA e RTDIT ficassem exatamente iguais. Também buscou-se reduzir a passagem de objetos como parâmetros de métodos, uma vez que a ferramenta CAT não conseguia adaptar o código corretamente nestas situações, levando com que estes objetos passados como parâmetros fossem substituídos pelos seus atributos, o que permitiu um aumento no número de parâmetros enviados em cada método. Também alterou-se o tipo dos atributos e das variáveis que eram *double* passando a serem *int*, tendo em vista que

o FemtoJava não suporta variáveis do tipo *double*. Essas alterações nos tipos das variáveis fez com que vários trechos de código que faziam referência aos tipos *double* fossem modificados para valores do tipo *int*, o que levou a uma mudança no valor do *checksum* final. Entretanto, a alteração mais importante nas versões foi a eliminação do paralelismo uma vez que a versão original do RayTracer é paralela através da implementação da interface Runnable. Como a ferramenta CAT não reconhece interfaces, foi necessário eliminar esta interface o que levou a eliminação do paralelismo nas versões adaptadas.

Terminada a etapa de modificação das versões para executar na CAT, o próximo passo foi a execução desta ferramenta sob estas versões com o objetivo de gerar novas versões que possuem chamadas de sub-rotinas buscando tratar de operações como instância de classes, acesso a atributos e métodos, entre outros. Estas chamadas de sub-rotinas são implementadas na DMML (*Dinamic Memory Management Library*), já explicada anteriormente, dentro da classe OpBase.java, inserida em cada uma das versões geradas pela CAT. Como um passo seguinte, começou a ser feita a execução de cada uma das versões geradas pela CAT com o intuito de observar se o valor final do *checksum* era o mesmo calculado pelas versões do RayTracer adaptadas. Entretanto, observou-se um erro durante as execuções de todas as versões relacionadas com falha de memória, fazendo com que nenhuma das versões conseguisse terminar de executar todas as iterações de seus *loops*. Estas iterações correspondem a varredura feita pelo RayTracer nos pixels contidos na tela, sendo que para cada um destes pixels é feito o mesmo cálculo para encontrar a sua cor no padrão RGB. A solução encontrada para resolver este problema foi diminuir o número de iterações, realizando testes com uma, duas e três iterações para cada uma das versões, observando-se em paralelo se ocorreria falhas quanto a falta de memória. Percebeu-se que para um número de iterações maiores do que três ocorria falta de memória para a execução. Desta forma, optou-se por utilizar um número de iterações menores do que quatro, buscando-se coletar o consumo de memória para uma, duas e três iterações.

Para a coleta dos dados de memória, foi utilizado inicialmente um *profiler* (NEVES, 2005) que contabilizava em uma variável o número de alocações feitas durante a execução da aplicação. Este *profiler* monitora as operações executadas dentro da DMML para verificar o instante em que ocorre a alocação ou desalocação de objetos no vetor da *heap*, contida na DMML. Para cada instante em que a operação correspondente com a alocação de objetos está sendo executada pela DMML, o tamanho do objeto é calculado pela mesma, sendo que este resultado é incrementado na variável do *profiler*. Esta variável contém a informação que representa a quantidade de memória que está sendo usada em um determinado instante. Quando ocorrem desalocações, o mesmo processo é feito, ou seja, a DMML calcula o tamanho do objeto no momento em que a desalocação está sendo executada e este valor é decrementado na variável do *profiler*. Assim,

é possível descobrir o consumo máximo de memória que uma determinada aplicação consome, inserindo-se a classe Profile.java, que descreve o *profiler*, junto as demais classes da aplicação (incluindo a classe OpBase). Entretanto, observou-se durante os experimentos que os resultados encontrados para o consumo de memória para cada uma das versões do RayTracer estavam com uma certa incoerência para as três iteração analisadas. Percebeu-se, através da depuração das versões instrumentadas pela CAT, que em certos momentos os valores que eram usados para referenciar as classes da aplicação estavam sendo confundidos ou trocados por valores armazenados pelas variáveis da classe gerados durante a execução da aplicação. Dessa forma, optou-se por não utilizar a classe Profile.java para a coleta dos dados de memória. Sendo assim, o recurso utilizado para obter o consumo de memória foi a classe OpBase.java, através do controle do tamanho do vetor referente a *heap*. Para cada uma das versões, buscou-se reduzir ao máximo o tamanho deste vetor sem que ocorresse erros durante a execução, coletando os resultados para uma, duas e três iterações. A Tabela 10, logo abaixo, apresenta os resultados de memória encontrado para cada uma das versões, com excessão da versão RTCeCa que não passou pela etapa de adaptação de código para a execução da CAT devido ao fato de que sua implementação está totalmente igual ao RTOriginal (sendo que a única diferença entre ambas é que na versão RTCeCa cada classe está contida em um pacote), o que não produz impactos sobre o consumo de memória da aplicação considerando o fluxo de execução adotado neste trabalho. Desta forma, estimou-se que os dados de memória encontrados para a versão RTOriginal são os mesmos para a versão RTCeCa. Os valores presentes na Tabela 10 estão representados em *Bytes*.

Tabela 10: Resultados para o consumo de memória

Versões	1 iteração	2 iterações	3 iterações
RTA	1,4 x 10^6	3,8 x 10^6	6,7 x 10^6
RTDIT	1,4 x 10^6	3,8 x 10^6	6,7 x 10^6
RTLCOM	6,1 x 10^6	12,2 x 10^6	18,3 x 10^6
RTNOM	6,1 x 10^6	12,2 x 10^6	18,3 x 10^6
RTNOP	3,8 x 10^6	7,6 x 10^6	11,4 x 10^6
RTNOSM	2,8 x 10^6	5,7 x 10^6	8,6 x 10^6
RTNOSA	104,9 x 10^6	209,8 x 10^6	314,7 x 10^6
RTOriginal	2,8 x 10^6	5,7 x 10^6	8,6 x 10^6

Percebe-se, a partir de uma observação na Tabela 10 que os dados de memória coletados para cada uma das versões do *RayTracer* apresentam um comportamento lógico. Inicialmente, verifica-se que a versão RTOriginal e a versão RTNOSM apresentam resultados totalmente iguais para as 3 iterações analisadas. Isso se deve ao fato de ambas implementações possuírem o mesmo número de classes assim como o mesmo número de atributos (assumindo que

o número de objetos alocados é o mesmo em cada aplicação). A única diferença entre ambas versões esta relacionada com os métodos, uma vez que alguns métodos da versão RTOriginal foram implementados de forma estática na versão RTNOSM, não havendo alterações no número de atributos e nem em seus tipos de dados. Dessa forma, não houve qualquer mudança quanto ao tamanho dos objetos o que implica que ambas versões deveriam apresentar os mesmos resultados de consumo de memória como foi demonstrado na Tabela 10.

Uma outra observação que se faz a partir da Tabela 10 é que as versões RTA e RTDIT obtiveram resultados para os dados de memória menores em relação aos encontrados pela versão original. O motivo para isso é que na versão adaptada do RTOriginal para executar a ferramenta CAT, eliminou-se todas as heranças contidas na aplicação. Entretanto, retirou-se apenas os modificadores de classe tais como *extends* e *abstract*, mantendo as classes pai e filho na versão desenvolvida para a CAT, com uma solução de implementação que busca substituir o comportamento encontrado entre uma classe herdada e uma classe que herda. Desta forma, nenhuma classe, com excessão das *interfaces*, foi excluída desta nova aplicação. Por outro lado, nas versões RTA e RTDIT que são totalmente iguais em código e que foram desenvolvidas a partir da versão adaptada do RTOriginal, buscou-se eliminar as classes que se comportavam como pais como forma de manter semelhantes o número de classes das versões originais do RTA e do RTDIT com as versões RTA e RTDIT adaptadas para serem executadas pela CAT. Para que a eliminação destas classes ocorressem, teve-se que transferir todas as variáveis de classe e os métodos da classe pai para a classe filho. Com isso, o fato destas duas versões terem obtido dados de memória menores do que a versão original pode estar relacionado com a quantidade de classes, tendo em vista que a versão RTOriginal possui mais classes para serem instanciadas do que as versões RTA e RTDIT.

Analisando novamente os resultados encontrados para os dados de memória, percebe-se que as versões RTLCOM e RTNOM possuem os mesmos valores para cada iteração analisada. Novamente, isso deve-se principalmente pelo fato de que ambas versões possuem os mesmos atributos, tanto em quantidade como em tipos de dados, o que não alteraria o tamanho dos objetos, levando a resultados iguais de dados de memória entre estas duas implementações. Do ponto de vista da modelagem da aplicação, a única diferença entre ambas versões é que na versão RTNOM existem menos métodos em relação a versão RTLCOM. Entretanto, verificou-se que estas duas versões apresentaram resultados maiores do que a versão RTOriginal. Após ser feita uma análise entre estas três implementações, constatou-se que as versões RTLCOM e RTNOM possuíam apenas quatro classes cada uma enquanto que a versão original possuía dezessete classes, levando a crer que nas duas implementações anteriores houve uma aglomeração de classes em relação ao RTOriginal. Isso gerou um aumento no número de campos em cada

uma das quatro classes destas duas versões (RTLCOM e RTNOM). Verificou-se também que uma das classes mais instanciadas na versão original, neste caso a classe Vec, possuía apenas três variáveis de classe. No momento em que as versões RTLCOM e RTNOM foram implementadas, essa mesma classe Vec acabou sendo fundida com outras classes formando uma nova classe que agora seria composta por atributos de cada uma das classes unidas. Essa nova classe formada para as versões RTLCOM e RTNOM é instanciada no mesmo número de vezes que a classe Vec da versão original. Entretanto, esta nova classe gera objetos maiores que os objetos criados a partir da classe Vec da versão RTOriginal, fazendo com que as duas versões derivadas da versão original consumam mais memória do que a versão original, como mostrado na Tabela 10.

Também é possível verificar na Tabela 10 que os resultados encontrados para a versão RTNOP são um pouco maiores do que os resultados para a versão original. A diferença entre estas duas implementações esta na redução do número médio de parâmetros recebidos por métodos na versão RTNOP. No entanto, para reduzir o número de parâmetros enviados para um método de uma determinada classe teve-se que criar novos atributos nesta mesma classe equivalentes a cada um dos parâmetros que buscou-se reduzir. Isso leva a um certo aumento no tamanho dos objetos gerados a partir destas classes modificadas em relação à versão RTOriginal, o que permite um maior consumo de memória pelo RTNOP do que a versão original. Entretanto, esta versão RTNOP apresentou dados de memória com valores menores do que as versões RTLCOM e RTNOM. Constatou-se que a classe Vec da versão RTNOP, que como foi dito anteriormente é uma das classes mais instanciadas da versão original assim como na versão RTNOP, não houve alterações em seus atributos, o que geraria um objeto com tamanho menor do que os objetos gerados a partir de classes criadas pela junção de outras classes nas versões RTLCOM e RTNOM, o que explica o motivo destas duas implementações possuírem consumo de memória maiores do que a versão RTNOP.

A coleta dos dados de memória para a versão RTNOSA foi feita de uma forma diferente em relação as demais versões. Primeiramente, fez-se uma análise do melhor caso para esta versão onde buscou-se determinar manualmente o tamanho de cada um dos objetos alocados estaticamente a partir das classes desta aplicação. Esta análise foi feita com o objetivo de ter conhecimento do consumo de memória ideal para esta versão, tendo em vista que todos os objetos desta aplicação são alocados na memória em tempo de compilação, sem que hajam objetos alocados ou desalocados durante a execução da versão. Com isso, o valor encontrado manualmente para o consumo de memória ideal foi de 10,1 x 10^3 *bytes*. Em seguida, foi feita uma análise do pior caso desta versão, onde utilizou-se a versão adaptada pela CAT do RTOriginal como base e o *profiler* discutido acima. Desta forma, foi feita uma estimação onde o total de

memória alocada (gerado pelo *profiler*) para a versão RTOriginal representaria o número limite de objetos alocados estaticamente pela versão RTNOSA. Assim sendo, encontrou-se valores para as três iterações analisadas, conforme apresentados na Tabela 10.

4.4.5 Coleta das métricas de eficiência para as versões instrumentadas

Antes de realizar a análise dos resultados, buscou-se conhecer os dados de eficiência que cada uma das versões instrumentadas pela CAT consumiam, com o objetivo de verificar se estes dados possuíam uma lógica semelhante com os dados encontrados para a Tabela 9. A Tabela 11 apresenta os valores coletados para o consumo de métricas de eficiência para cada uma das versões instrumentadas. Novamente, a versão RTCeCa que não passou pela etapa de adaptação de código para a execução da ferramenta CAT por possuir a mesma implementação da versão original, não é mostrada na Tabela 11. É importante salientar que estes valores foram coletados usando-se a ferramenta BIT.

Tabela 11: Métricas de eficiência para as versões instrumentadas.

Versões	Ciclos (Total)	Energia (*nJ*)	Instruções (Total)
RTOriginal	7,13 x 10^{11}	3,50 x 10^{11}	5,35 x 10^{11}
RTA	6,33 x 10^{11}	3,11 x 10^{11}	4,78 x 10^{11}
RTDIT	6,33 x 10^{11}	3,11 x 10^{11}	4,78 x 10^{11}
RTLCOM	5,52 x 10^{11}	2,74 x 10^{11}	4,36 x 10^{11}
RTNOM	5,22 x 10^{11}	2,61 x 10^{11}	4,19 x 10^{11}
RTNOP	12,63 x 10^{11}	6,22 x 10^{11}	9,69 x 10^{11}
RTNOSM	9,54 x 10^{11}	4,67 x 10^{11}	7,12 x 10^{11}
RTNOSA	1,31 x 10^{11}	0,58 x 10^{11}	0,77 x 10^{11}

Os valores contidos na Tabela 11 correspondem a execução de apenas três iterações. Todas as versões foram executadas com uma única *thread*, neste caso a *thread* correspondente ao método estático *main*, devido a motivos citados na seção anterior. Comparando os valores obtidos na Tabela 11 com os valores obtidos na Tabela 9, percebe-se de imediato que os resultados são de certa forma aproximados. Como prova disso, as versões RTA e RTDIT continuam com valores totalmente iguais para as métricas de energia, número de ciclos e número de instruções executadas. O motivo destes valores terem sido iguais para estas duas implementações geradas pela CAT é que ambas versões estão totalmente iguais, ou seja as duas versões possuem as mesmas classes e as mesmas estruturas que compõem cada classe, como já foi mencionado na seção anterior. Outra semelhança entre as duas tabelas em questão é que a versão RTNOM é a versão que menos consome energia e que possui o número de ciclos menor do que as demais, com excessão da versão RTNOSA que na Tabela 11 obteve dados de eficiência ainda menores do que a RTNOM. Entretanto, algumas diferenças podem ser percebidas nos valores de cada uma destas

tabelas. Uma destas diferenças é que na Tabela 9 a versão RTNOSA foi a que mais consumiu energia e ciclos em relação as demais versões enquanto que na Tabela 11 a versão com maiores resultados dentre todas foi a RTNOP. Esta mesma versão RTNOSA, que na Tabela 9 apresenta os maiores valores em relação as demais, contém na Tabela 11 os menores valores dentre todas as versões instrumentadas. Uma outra diferença encontrada está na versão RTNOSM que na Tabela 9 contém resultados iguais às versões RTOriginal, RTA, RTDIT e RTLCOM enquanto que na Tabela 11 contém valores maiores que estas versões.

Uma explicação para tais diferenças encontradas pode estar relacionado com o Gerenciamento Dinâmico de Memória, uma vez que na Tabela 9 não foram levantados os valores reais, como consumo de energia e quantidade de ciclos necessários, que a Máquina Virtual Java levaria para alocar, desalocar e acessar um objeto. Já na Tabela 11, cada versão instrumentada pela CAT possuem estes valores reais, ou seja, estão sendo incluídos a quantidade de energia e a quantidade de ciclos que são necessárias para a alocação e desalocação de objetos. Em outras palavras, a ferramenta CAT realiza a substituição dos *bytecodes* de gerenciamento dinâmico de memória por chamadas a métodos estáticos (suportados pelo FemtoJava) que implementam o comportamento destas instruções. Estes métodos estáticos são implementados na classe OpBase.java. Esta classe possui um método cuja implementação é capaz de efetuar a devida alocação de um objeto na memória *heap*, a partir da análise do *bytecode new* disponível no arquivo *.jad*, conforme citado anteriormente. Desta forma, quando foi feita a execução da ferramenta BIT sob estas versões instrumentadas pela CAT, os *bytecodes new*, referentes a alocação de um determinado objeto, passaram a ser contabilizados nos demais valores de energia e ciclos, uma vez que estão sendo acrescentados os resultados referentes a cada um dos *bytecodes* presentes no método que trata a operação *new*.

4.4.6 Análise dos dados de energia e ciclos

Após serem coletados os dados de eficiência para cada uma das versões instrumentadas pela CAT, realizou-se a análise sob os resultados encontrados na Tabela 9. Optou-se por realizar a análise dos resultados de eficiência nas versões originais do RayTracer (versões não adaptadas pela CAT) por possuírem um código mais limpo e mais claro para o devido entendimento em relação aos códigos gerados pela CAT. Abaixo, são listadas todas as versões usadas neste trabalho apresentando para cada uma justificativas para os resultados encontrados quanto ao consumo de energia e o número de ciclos. É importante salientar, como foi dito anteriormente, que estas versões, com excessão da versão RTNOSA, não foram desenvolvidas durante este trabalho.

- Versão RTA

Nesta versão, buscou-se eliminar todo o nível de abstração que a versão original possuía. Desta forma, eliminou-se todas as classes abstratas e interfaces implementadas no RTOriginal, tirando a capacidade de qualquer classe estender ou implementar uma outra. A única interface implementada nesta versão é a Runnable, interface esta que contém o método *run* que deve ser definido pela classe que implementa esta interface, sendo que este método é capaz de ser executado por uma ou mais *threads* em paralelo. Para que ocorresse a redução do número de classes abstratas, teve-se que transferir todas as variáveis e métodos implementados na classe abstrata para as classes que as estendem, implementado assim todos os métodos abstratos nestas últimas classes. Para a eliminação das interfaces da aplicação, o processo foi mais simples. Neste caso, como a versão original continha somente a interface Serializable, além da interface Runnable, e como esta interface também não contém nenhum método a ser definido, apenas buscou-se eliminar esta interface da aplicação e tirar a capacidade de algumas classes de RTA de implementarem esta interface.

Assim sendo, percebeu-se que não houve modificações nas linhas de código Java, o que implica que tanto a versão RTA como a versão original do RayTracer terem os mesmos *bytecodes* para cada linha de código em alto-nível (código Java). Este é o motivo pelo qual estas duas versões terem os mesmos resultados para o consumo energético e para a quantidade de ciclos.

- Versão RTDIT

Esta versão foi implementada buscando reduzir a profundidade de herança. A exemplo disso, a implementação desta versão seguiu praticamente os mesmos passos para a geração da versão RTA. Entretanto, na geração da versão RTDIT, buscou-se apenas eliminar todas as classes abstratas e todas as classes que continham subclasses na aplicação original (RTOriginal). Desta forma, a interface Serializable foi mantida nesta versão, assim como a interface Runnable. Para eliminar as heranças na versão RTDIT, foi necessário trazer todas as variáveis e métodos da classe herdada para a classe que herda, redirecionando toda e qualquer referência existente para a classe eliminada para a classe que herda.

Desta forma, assim como ocorreu com a versão RTA, não houve nenhuma modificação em nível de código java em RTDIT. Isso fez com que os *bytecodes* relacionados com cada linha de código em alto-nível para esta versão fossem exatamente os mesmos *bytecodes* presentes para cada linha de código Java da versão RTOriginal. Isso permitiu com que os valores gerados para o consumo de energia e o número de ciclos para esta versão fossem iguais ao resultados obtidos pelas versões RTOriginal e RTA. É importante notar que ao se reduzir a profundidade de herança, pode também estar sendo reduzido a abstração, uma vez que as classes herdadas eliminadas podem ser abstratas.

- Versão RTCeCa

Esta versão foi desenvolvida sem que houvesse alterações em nível de código e de classes quando relacionada com a versão original. A única diferença entre estas duas versões é que na RTCeCa, cada uma das classes da aplicação foi inserida em um pacote específico para verificar se haveria algum impacto em termos de eficiência quanto ao relacionamento de classes em pacotes distintos. Observou-se que os dados de energia e de ciclos para esta versão são os mesmos em relação à versão original, tendo em vista que não houve alterações em nível de código Java, não havendo também mudanças nos *bytecodes* para cada uma das instruções destas duas aplicações. Desta forma, os custos de energia e ciclos para executar cada um dos *bytecodes* da versão RTOriginal são os mesmos custos encontrados para a versão RTCeCa.

- Versão RTLCOM

Esta versão foi desenvolvida com o intuito de reduzir a coesão ao máximo para que seja possível observar os resultados gerados para o consumo de energia e de ciclos. Desta forma, buscou-se aglutinar todas as classes que possuíam uma certa relação em uma única classe, gerando assim quatro classes ao todo nesta implementação. Esta transformação permitiu reduzir a profundidade de herança, uma vez que eliminou-se todas as classes herdadas e abstratas, transferindo todos os atributos e métodos destas classes para uma única classe. Uma outra alteração percebida foi o aumento do número de construtores por classe, uma vez que com a aglomeração de classes em uma só, os construtores de todas as classes eliminadas passaram a ser transferidos para a classe em questão.

Desta forma, após ser feita a análise dos *bytecodes* desta versão, percebeu-se que não houve nenhuma alteração quando comparados com os *bytecodes* encontrados para a versão original. O motivo para isso é que não houve nenhuma mudança em linha de código Java entre uma versão e a outra. Assim sendo, os resultados para as métricas de eficiência desta versão serem iguais aos valores encontrados para o RTOriginal são justificados pelo fato de não ter ocorrido nenhuma alteração entre os *bytecodes* destas duas versões.

- Versão RTNOM

Esta versão foi desenvolvida a partir da versão RTLCOM, possuindo ao final da implementação as mesmas quatro classes da versão anterior. Na versão RTNOM, buscou-se reduzir ao máximo o número de métodos da aplicação. Para isso, observou-se os pontos onde ocorriam

as chamadas de métodos, substituíndo em seguida estas chamadas pelos corpos dos métodos chamados.

Comparando esta versão com o RTOriginal, em nível de *bytecodes*, observou-se que muitas chamadas de métodos deixaram de ser executadas. Por este motivo, esta versão obteve dados de energia e de ciclos menores que os obtidos pela versão original. Neste caso, o *bytecode* relacionado com a chamada de métodos é o *invokevirtual* que consome 34003 como valor parcial para a energia e precisa de doze ciclos para ser executado. Com a diminuição das chamadas de métodos na versão RTNOM, estes valores passaram a ser o diferencial entre esta versão e a versão original.

Abaixo, segue um exemplo de um trecho de código que ocorreu entre estas duas versões, mostrando a troca da chamada do método na versão original pelo corpo deste mesmo método na versão RTNOM.

```
/*
*   Código 5
*/
public void intersect() {
 Vec a = new Vec(5);
 Vec b = new Vec(10);
 v.sub(a, b);
// 0  0:aload_0
// 1  1:getfield        #29  <Field Vec v>
// 2  4:aload_0
// 3  5:getfield        #20  <Field Vec a>
// 4  8:aload_1
// 5  9:getfield        #41  <Field Vec b>
// 6 12:invokevirtual   #46  <Method void Vec.sub(Vec, Vec)>
}

public void sub(Vec a, Vec b) {
 x = a.x - b.x;
// 0  0:aload_0
// 1  1:aload_1
// 2  2:getfield        #17  <Field double x>
// 3  5:aload_2
// 4  6:getfield        #17  <Field double x>
// 5  9:dsub
```

```
// 6 10:putfield        #17  <Field double x>
}
```

Nestes exemplos de código, que ocorreram na versão RTOriginal, *v* é um atributo do tipo *Vec* que chama o método *sub* passando como parâmetros as variáveis *a* e *b*. No método *sub* é realizada a subtração dos campos de *a* e *b* e estes valores são armazenados no campo *x* do atributo *v*. Em destaque, são apresentados os *bytecodes* para as duas linhas de código Java acima. Para este pequeno exemplo, a aplicação consumiria 105413 *nJ* de energia e levaria 24 ciclos para ser executado. Abaixo, segue um trecho de código da versão RTNOM onde houve a substituição da chamada de método pelo corpo do método.

```
/*
*   Código 6
*/
public void intersect() {
 Vec a = new Vec(5);
 Vec b = new Vec(10);
 v.x = a.x - b.x;
// 0  0:aload_0
// 1  1:getfield        #42  <Field TournamentBarrier v>
// 2  4:aload_0
// 3  5:getfield        #29  <Field TournamentBarrier a>
// 4  8:getfield        #56  <Field double TournamentBarrier.x>
// 5 11:aload_1
// 6 12:getfield        #59  <Field TournamentBarrier b>
// 7 15:getfield        #56  <Field double TournamentBarrier.x>
// 8 8:dsub
// 9 19:putfield        #56  <Field double TournamentBarrier.x>
}
```

Para este trecho de código acima, a aplicação consumiria 59751 *nJ* de energia e levaria 10 ciclos para ser executado, valores estes menores em relação aos encontrados no exemplo da versão original, uma vez que o *bytecode invokevirtual* não está sendo executado.

- Versão RTNOSM

Nesta versão, buscou-se aumentar o número de métodos estáticos por classe. Para isso, teve-se que acrescentar o modificador *static* a frente da maioria dos métodos da aplicação. Essa

transformação permitiu com que os métodos modificados fossem chamados a partir de suas classes e não mais de um objeto. Estes objetos passaram a ser passados como parâmetros para que possam ser acessados internamente ao corpo dos métodos estáticos.

Estas transformações não influenciaram nos resultados obtidos para esta versão, uma vez que foram produzidos exatamente os mesmo valores obtidos pela versão original. Verificou-se, em nível de *bytecodes*, que a única alteração ocorrida entre esta versão e a RTOriginal foi a troca do *bytecode invokevirtual*, responsável por realizar a chamada de um método comum, pelo *bytecode invokestatic* que invoca os métodos estáticos da aplicação. Estes dois *bytecodes* possuem os mesmos valores para o consumo de energia e o número de ciclos, o que demonstra que estas duas versões obtiveram os mesmos resultados para estas duas métricas de eficiência.

Abaixo, segue um exemplo contendo linhas de código Java encontrados entre estas duas versões, mostrando os *bytecodes* utilizados para cada linha de código.

```
/*
*   Versão RTOriginal
*/
D.normalize();
// 24 50:aload_0
// 25 51:getfield        #30  <Field Vec D>
// 26 54:invokevirtual   #32  <Method double Vec.normalize()>
// 27 57:pop2
// 28 58:return
```

```
/*
*   Versão RTNOSM
*/
Vec.normalize(D);
// 24 50:aload_0
// 25 51:getfield        #30  <Field Vec D>
// 26 54:invokestatic    #32  <Method double Vec.normalize(Vec)>
// 27 57:pop2
// 28 58:return
```

Neste exemplo, a linha de código que representa a versão original possui uma variável de classe do tipo *Vec* que invoca o método *normalize* enquanto que a linha que representa a versão RTNOSM realiza a chamada do método estático *normalize* a partir de sua classe *Vec* enviando como parâmetro a variável *D*. Deve-se passar esta variável de classe como parâmetro para que o objeto *D* possa ser acessado internamente ao método estático. Comparando-se os

bytecodes destas duas linhas de código, verifica-se que a única alteração entre ambas foi a troca do *bytecode invokevirtual* pelo *invokestatic*.

- Versão RTNOP

Esta versão foi desenvolvida com o objetivo de reduzir o número de parâmetros enviados por métodos. Para que esta implementação ocorresse, teve-se que acrescentar novos campos (variáveis de classe) em uma determinada classe relacionados com cada parâmetro eliminado nos métodos pertencente a esta classe. Assim, antes do método modificado ser chamado, essas variáveis de classe devem receber os valores que os parâmetros eliminados receberiam para que possam ser usadas internamente à este método.

Abaixo, segue um exemplo encontrado entre estas duas versões demonstrando as alterações ocorridas entre a linha de código pertencente à versão original e a linha Java pertencente à versão RTNOP.

```
/*
*   Versão RTOriginal
*/
render(inter);
// 23 45:aload_0
// 24 46:aload_1
// 25 47:invokevirtual   #69  <Method void render(Interval)>
```

```
/*
*   Versão RTNOP
*/
this.inter = inter;
// 23 45:aload_0
// 24 46:aload_1
// 25 47:putfield        #71  <Field Interval inter>
render();
// 26 50:aload_0
// 27 51:invokevirtual   #75  <Method void render()>
```

Percebe-se que houve algumas alterações em nível de *bytecodes* para este exemplo. No primeiro caso, correspondente à versão original, o método *render* é invocado passando como parâmetro o objeto *inter*. Verifica-se que apenas três *bytecodes* são necessários para executar esta instrução: *aload_0* para invocar o objeto referente ao método *render*, *aload_1* para invocar

o objeto *inter* e *invokevirtual* para realizar a chamada do método *render*. Para estes três *bytecodes*, o consumo energético seria 41794 *nJ* e levaria 14 ciclos para serem executados. No segundo caso, o método *render* é chamado para ser executado sem que ocorra passagem de parâmetros. Assim sendo, é criado o campo *inter* na classe a qual o método *render* pertence, sendo inicializado antes deste método ser invocado. Observa-se que alguns *bytecodes* tiveram que ser inseridos em relação à versão original, surgindo três *bytecodes* relacionados com a linha onde o campo *inter* é preparado para acesso interno no escopo do método *render*: *aload_0* para carregar o objeto que possui o campo *inter*, *aload_1* para carregar o objeto *inter* e *putfield* para transferir o valor contido no objeto *inter* para o campo *inter*. Também surgiram dois *bytecodes* para a linha Java que chama o método *render*: *aload_0* para carregar o objeto referente ao método em questão e *invokevirtual* para realizar a chamada do método *render*. Para este exemplo da versão RTNOP, estes cinco *bytecodes* consumiriam 53050 *nJ* de energia e levaria 16 ciclos para ser executada, ou seja, seriam valores maiores aos encontrados na versão original. Este motivo justifica o fato da versão RTNOP ter gerado valores maiores em relação aos valores da versão RTOriginal.

- Versão RTNOSA

Para o desenvolvimento da versão RTNOSA onde buscou-se aumentar ao máximo o número de atributos estáticos, muitas alterações tiveram que ser feitas em relação a versão RTOriginal. A principal alteração esta justamente na forma como os atributos estáticos foram implementados. Foram utilizadas matrizes em cada um dos atributos estáticos onde o número de colunas destas matrizes correspondem ao número de *threads* que estão sendo executadas por esta aplicação. O motivo pelo qual foi usada esta estratégia esta abordado na Seção 4.4.3.

Observou-se que estas modificações levaram a resultados maiores quanto ao consumo energético e o número de ciclos em relação a versão original. Abaixo, segue o exemplo de um método buscando comparar os *bytecodes* relacionados com o acesso a um atributo simples, presentes na versão RTOriginal, com o acesso a um atributo estático, presente na versão RTNOSA.

```
1 /*
2 *   Versão RTOriginal
3 */
4 public void scale(double t) {
5  x *= t;
6 // 0 0:aload_0
7 // 1 1:dup
8 // 2 2:getfield        #17  <Field double x>
```

```
// 3 5:dload_1
// 4 6:dmul
// 5 7:putfield        #17  <Field double x>
}
```

```
/*
*   Versão RTNOSA
*/
public static void scale(int n, int id, double t) {
 x[n][id] *= t;
//  0  0:getstatic       #22  <Field double[][] x>
//  1  3:iload_0
//  2  4:aaload
//  3  5:iload_1
//  4  6:dup2
//  5  7:daload
//  6  8:dload_2
//  7  9:dmul
//  8 10:dastore
}
```

Desta forma, no código referente à versão original, a instrução presente no método *scale* busca acessar um atributo simples (x) para realizar um cálculo matemático com a variável passada como parâmetro (t). Verifica-se que os *bytecodes* utilizados para executar esta instrução consomem 33218 *nJ* de energia e levam 9 ciclos para ser executada. Por outro lado, no código da versão RTNOSA, a instrução do método *scale* busca acessar um atributo estático (x) através dos indexadores n e id para realizar o mesmo cálculo matemático anterior com a variável passada como parâmetro (t). Conforme dito anteriormente, cada atributo estático foi implementado utilizando matrizes. Assim sendo, os indexadores passados como parâmetro no método *scale* da versão RTNOSA servem para localizar a posição ideal para armazenar um determinado valor na matriz deste atributo, sendo que a variável n é utilizada para referenciar a linha desta matriz, localizando a posição exata do campo ao qual se quer ter acesso enquanto que a variável id seleciona a coluna referente ao número da *thread* que esta sendo executada. Desta forma, observa-se que os *bytecodes* necessários para executar a instrução do método *scale* consomem 66895 *nJ* e levam 28 ciclos para ser executada. Com isso, estes exemplos que comparam instruções de acesso a atributos simples e estáticos justificam os dados de eficiência encontrados para a versão RTOriginal e RTNOSA na Tabela 9.

4.4.7 Comparação dos resultados

A Tabela 12 apresenta um comparativo entre as variações das métricas de qualidade de software, de energia, de número de ciclos e do consumo de memória para cada versão desenvolvida do RayTracer. Os dados de memória mostrados nesta tabela representam a execução de 3 iterações em cada uma das versões. Todas as versões foram desenvolvidas a partir do RTOriginal, com excessão da versão RTNOM que foi desenvolvida tendo como ponto de partida a versão RTLCOM, como pode ser visto na primeira coluna da Tabela 12.

Tabela 12: Comparação dos resultados obtidos.

Versões	Métrica OO	Energia	Ciclos	Memória
$RTOriginal \rightarrow RTA$	100%	0%	0%	22,1%
$RTOriginal \rightarrow RTDIT$	17,36%	0%	0%	22,1%
$RTOriginal \rightarrow RTCeCa$	42,86%	0%	0%	0%
$RTOriginal \rightarrow RTLCOM$	56,06%	0%	0%	53,01%
$RTLCOM \rightarrow RTNOM$	47,37%	34,83%	44,17%	53,01%
$RTOriginal \rightarrow RTNOP$	29,68%	4,28%	3,62%	24,57%
$RTOriginal \rightarrow RTNOSM$	76,93%	0%	0%	0%
$RTOriginal \rightarrow RTNOSA$	93,7%	43,15%	48,72%	97,27%

Nesta tabela, cada valor representa a diferença percentual encontrada entre a versão desenvolvida (a direita da seta na primeira coluna) e a versão tomada como ponto de partida (a direita da seta). Desta forma, observa-se que a variação de 100% na métrica de abstração não causa impactos no consumo de energia e no número de ciclos. Entretanto, a variação nesta métrica provocou impactos no consumo de memória, onde verificou-se que ocorreu uma diferença de 22,1% entre as versões RTA e RTOriginal, sendo que a versão original consome mais memória do que a versão RTA. É possível analisar também que diminuindo a profundidade de herança, provocada pela variação de 17,36%, não gera impactos nos dados de energia e de ciclos. Contudo, com a diminuição na profundidade de herança ocorre a variação de 22,1% nos dados de memória. Observa-se também, na Tabela 12, que variando-se 42,86% nas métricas Ca e Ce e 76,93% na métrica NOSM não provoca impactos nos dados de energia, ciclos e memória. Diminuido-se a coesão nos métodos, através da variação de 56,06%, gera impactos apenas nos dados de memória, onde verificou-se uma diferença de 53,01%, sendo que a versão RTLCOM consome mais memória do que a versão original.

Com isso, a análise na Tabela 12 permite concluir que apenas as variações nas métricas NOM, NOP e NOSA causam impactos em todas as métricas de eficiência analisadas neste trabalho. Assim sendo, diminuindo-se 47,37% do número de métodos causa 34,83% no consumo de energia e 44,17% no número de ciclos, sendo que a versão RTNOM é a que menos consome

dados de energia e de ciclos entre todas as versões. Esta variação também causou impactos nos dados de memória, onde verifica-se que houve uma diferença percentual de 53,01%, tendo em vista que a versão RTNOM consome mais dados de memória do que a versão original. Diminuindo o número de parâmetros enviados por métodos, representado na Tabela 12 pela variação de 29,68%, provoca variações de 4,28% nos dados de energia, 3,62% nos dados de ciclos e 24,57% no consumo de memória, sendo que a versão RTNOP possui valores para os dados de energia, ciclos e memória maiores do que a versão original. Por fim, aumentando-se o número de atributos estáticos, com a diferença percentual de 93,7%, levou a variações de 43,15% no consumo de energia, 48,72% no número de ciclos e 97,27% nos dados de memória, uma vez que a versão RTNOSA consome mais energia, ciclos e memória do que a versão RTOriginal.

5 CONSIDERAÇÕES FINAIS

Este trabalho tratou principalmente da análise experimental para os dados de energia, ciclos e memória para cada uma das implementações desenvolvidas a partir da versão original do RayTracer, buscando verificar quais métricas de qualidade de software provocam impactos nas métricas de eficiência. Para isso, a primeira etapa deste trabalho foi desenvolver uma nova versão do RayTracer com o objetivo de aumentar o número de atributos estáticos como forma de analisar os impactos que esta variação poderia vir a causar sobre os dados de energia, ciclos e de memória. Optou-se, inicialmente, em desenvolver uma versão com uma única *thread* como forma de simplificar o desenvolvimento da aplicação e verificar se os resultados encontrados correspondiam com os resultados gerados pela versão original (RTOriginal). Desenvolveu-se, a seguir, uma versão onde buscou-se dar suporte ao paralelismo. Após o desenvolvimento da versão NOSA, utilizou-se a ferramenta de instrumentação BIT que permite realizar modificações no comportamento de uma aplicação gerando uma nova ferramenta (PAT) para fazer a análise dos custos energéticos e de ciclos. Esta ferramenta foi usada para coletar os dados de energia e de ciclos para a versão RTNOSA. Na próxima etapa, buscou-se coletar os dados de memória para todas as versões desenvolvidas a partir do RayTracer original. Assim sendo, optou-se em utilizar a ferramenta CAT que realiza uma adaptação de código como forma de substituir todos os *bytecodes* referentes ao gerenciamento dinâmico de memória (não suportados pelo Femto-Java) por chamadas de métodos estáticos que tratam da implementação comportamental destas instruções. Os dados de memória foram coletados para cada uma das versões por meio do controle da variável *heap* presente na DMML, inserida na aplicação pela ferramenta CAT. Após a coleta dos dados de eficiência, buscou-se analisar os resultados com o objetivo de encontrar justificativas pelos quais estes valores foram gerados.

Com isso, este trabalho permitiu concluir, por meio da análise dos dados de eficiência coletadas, que as únicas métricas de qualidade de software que provocaram impactos sobre o desempenho da aplicação foram: NOM, NOP e NOSA, tendo em vista que estas métricas foram as únicas que causaram interferência nos dados de energia, ciclos e de memória. Observou-se que a redução do número de métodos conduz a ganhos no consumo de energia e no número de ciclos levando, porém a um maior consumo de memória. Verificou-se também que aumentando o número de atributos estáticos ou diminuindo o número de parâmetros passados por métodos podem não ser boas práticas de programação uma vez que estas duas métricas geraram dados de eficiência maiores do que a versão original do RayTracer. Entretanto, para obter ganhos no consumo de memória, uma boa prática de programação poderia ser reduzir o nível de abstração ou reduzir a profundidade de herança, embora estas implementações não gerem impactos nos

dados de energia e ciclos.

A Tabela 13 apresenta todas as etapas que foram desenvolvidas durante este trabalho.

Tabela 13: Cronograma

Atividade	Semestre 1					Semestre 2				
	1	2	3	4	5	1	2	3	4	5
Estudo da aplicação original	●									
Desenvolvimento da versão NOSA	●	●	●							
Coleta das métricas de qualidade de software			●							
Coleta das métricas de eficiência				●						
Escrita da Monografia de TCC I				●	●					
Apresentação do TCC I					●					
Estudo detalhado de *profiles* de memória						●				
Coleta de consumo de memória						●	●			
Estudo de métodos de co-relacionamento							●	●		
Análise dos impactos								●		
Escrita da monografia de TCC II							●	●	●	●
Apresentação da monografia de TCC II										●

REFERÊNCIAS

AMD. *Advanced Micro Devices*. 2011. [Online; accessado em 17-Novembro-2011]. Disponível em: <www.amd.com/br>.

APPEL, A. Some techniques for shading machine renderings of solids. p. 37–45, 1968.

CARDOSO, B.; ROSA, S. R. A. dos S.; FERNANDES, T. M. *Multi-core*. 2011. [Online; accessado em 09-Novembro-2011]. Disponível em: <http://www.ic.unicamp.br/ rodolfo/Cursos/mc722/2s2005/Trabalho/g07-multicore.pdf>.

DEITEL, H. M.; DEITEL, P. J. *Java Como Programar*. [S.l.]: São Paulo, 2006. ISBN 978-85-7605-019-3.

DJ. *Decompiler Java*. 2011. [Online; accessado em 16-Novembro-2011]. Disponível em: <http://members.fortunecity.com/neshkov/dj.html>.

FOLEY, J. D. *Introduction to Computer Graphics*. [S.l.: s.n.], 1993. ISBN 0201609215.

FRANCISCO, P. S. *Reflexão das Ondas*. 2011. [Online; accessado em 21-Novembro-2011]. Disponível em: <http://www.portalsaofrancisco.com.br/alfa/mecanica-ondulatoria/reflexao-das-ondas.php>.

HENDERSON-SELLERS, B. *Object-Oriented Metrics, Measures of Complexity.* [S.l.: s.n.], 1996.

INTEL. *Intel Corporation*. 2011. [Online; accessado em 17-Novembro-2011]. Disponível em: <http://www.intel.com>.

ITO, S. A. *Projeto de Aplicações Específicas com Microcontroladores Java Dedicados*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul, 2000.

JP. *JProfiler - Java Profiler.* 2011. [Online; accessado em 18-Novembro-2011]. Disponível em: <http://www.ejtechnologies.com/products/jpro

ler/overview.html>.

LCOM. *Object-Oriented Metrics: LCOM*. 2011. [Online; accessado em 20-Dezembro-2011]. Disponível em: <http://www.computing.dcu.ie/ renaat/ca421/LCOM.html>.

LEE, H. B.; ZORN, B. G. *BIT: Bytecode Intrumenting Tool*. 2011. [Online; accessado em 19-Novembro-2011]. Disponível em: <http://www.cs.colorado.edu/~hanlee/BIT>.

MARTINS, E. Métricas oo. Março 2003.

METRICS. *Metrics 1.3.6 - Getting started*. 2011. [Online; accessado em 20-Novembro-2011]. Disponível em: <http://metrics.sourceforge.net/>.

MILLS, E. E. Software metrics. Software Engineering Institute/SEI, 1998.

NEVES, B. S. *Gerência Dinâmica de Memória em Aplicações Java Embarcadas*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul, 2005.

OLIVEIRA, M. F. S. et al. *Software quality metrics and their impact on embedded software*. 2011. [Online; accessado em 07-Novembro-2011]. Disponível em: <ftp://ftp.inf.ufrgs.br/pub/simoo/.../mompes08.pdf>.

SALES, P. da S. B.

Avaliação de Desempenho de Ferramentas de Renderização de Imagens em Clusters openMosix e Arquiteturas Multicore — Escola Politécnica de Pernambuco, 2008.

SCHACH, S. R. *Engenharia de Software: Os Paradigmas Clássico & Orientado a Objetos*. [S.l.]: São Paulo, 2009. ISBN 978-85-7726-045-4.

SILVA, F. W. S. V. da. *Introdução ao Ray Tracing*. 2011. [Online; accessado em 16-Novembro-2011]. Disponível em: <http://w3.impa.br/ nando/publ/rt/>.

SMITH, L. A.; BULL, J. M.; OBDRZáLEK, J. A parallel java grande benchmark suite. p. 8–8, 2001.

TI. *TI Expert.NET*. 2011. [Online; accessado em 07-Dezembro-2011]. Disponível em: <http://www.tiexpert.net/programacao/java/pacotes.php>.

VALCAR.NET. *Tutor de CG*. 2011. [Online; accessado em 22-Novembro-2011]. Disponível em: <http://www2.dem.inpe.br/val/homepage/tutor/index.html>.

WHITTED, T. An improved illumination model for shaded display. June 1980.

WIKIPEDIA. *Métrica de Software*. 2011. [Online; accessado em 13-Novembro-2011]. Disponível em: $<http://pt.wikipedia.org/wiki/M\%C3\%A9trica\_de\_software>$.

XENOS, M. et al. Object-oriented metrics – a survey. Proceedings of the FESMA 2000, 2000.

YJP. *Java Profiler: The profilers for Java and .NET professionals*. 2011. [Online; accessado em 18-Novembro-2011]. Disponível em: <http://www.yourkit.com/>.

ZHIRNOV, V. V. et al. Limits to binary logic switch scaling — a gedanken model. PROCEEDINGS OF THE IEEE, v. 91, p. 1934–1939, November 2003.