Projeto URBIS Amazônia

TERRAME *OBSERVER*: UM *PIPELINE* EXTENSÍVEL PARA VISUALIZAÇÃO EM TEMPO REAL DE MODELOS ESPACIALMENTE-EXPLÍCITOS

Relatório apresentado à Fundação de Ciência, Aplicações e Tecnologias Espaciais como instrumento de acompanhamento e avaliação dos componentes de visualização e ferramentas desenvolvidas no período de 01/01//2012 à 31/12//2013.

Bolsista: Antônio José da Cunha Rodrigues

Antônio José da Cunha Rodrigues

Assinatura do coordenador do projeto no INPE:

Dr. Antonio Miguel Vieira Monteiro

RESUMO

A visualização científica é uma eficiente ferramenta de síntese, pois permite transformar o grande volume de dados científicos produzidos diariamente em informações relevantes. Quando aplicada em áreas como a de modelagem ambiental, ela contribui em diversos níveis, como, no desenvolvimento e melhoria dos modelos ambientais, interpretação e comunicação de resultados e no apoio à tomada de decisão e à definição de políticas públicas. A visualização é o resultado de uma sequência de processos de transformações, chamada *pipeline*, na qual imagens bidimensionais são construídas a partir dos dados em estudo. Este relatório apresenta a concepção e o projeto de uma arquitetura de alto desempenho para *pipelines* destinados à visualização de simulações ambientais. Essa arquitetura chamada TerraME *Observer* foi implementada como uma extensão do simulador ambiental TerraME e avaliada segundo análise de desempenho e planejamento de capacidade. Esse trabalho obteve resultados na formação de recursos humanos: tutoria de alunos de Iniciação Científica e um título de mestrado, em publicações e em forma de resultados científicos: comparando-se os desempenhos das versões inicial e final da arquitetura, os resultados dos experimentos mostram uma redução de 60% a 80% no tempo de resposta do serviço de visualização e um aumento inferior a 7% no consumo de memória. A arquitetura TerraME *Observer* é extensível, flexível e pode ser utilizada por qualquer outro ambiente de modelagem para implementar seus serviços de visualização.

LISTA DE ILUSTRAÇÕES E FIGURAS

Figura 2.1 – Estágios do processo de visualização. Adaptado de (WOOD et al., 2005). 13

Figura 2.7 – Arquitetura da plataforma de modelagem TerraME. Adaptado de (CARNEIRO et al., 2013). 15

Figura 2.8 – Ambiente de execução do TerraME 16

Figura 2.20 – Padrão de projeto de *software Observer*. Adaptado de (GAMMA et al., 1995)............. 18

Figura 2.21 – Diversos visualizadores compartilham as informações mantidas no *BlackBoard*.......... 19

Figura 3.1 – Visão geral do *pipeline* de visualização do TerraME. Adaptado de Wood et al. (2005).. 21

Figura 3.2 – Simulação e visualizações podem residir em espaços de memória de processos distintos ou em camadas de software distintas. No TerraME, as variaveis de estado do modelo estão na pilha do interpretador Lua e as visualizações estão no *framework* C++. 22

Figura 3.3 – Estruturação do padrão *Observer* e do padrão *BlackBoard* no pipeline do TerraME 24

Figura 3.4 – Diagrama de classe das principais entidades envolvidas na comunicação entre modelo e visualização – Integração entre os padrões de projeto *Observer* e *Blackboard* 25

Figura 3.5 – Diagrama de sequência – Interações entre os padrões de projeto *Observer e Blackboard*26

Figura 3.6 – Detalhamento do escalonamento dos módulos 27

Figura 3.7 – Detalhamento da estrutura de dados que implementa o *BlackBoard*............................... 28

Figura 3.8 – Diagrama de classe do *BlackBoard* 29

Figura 3.9 – Formato de datagrama 30

Figura 3.10 – Máquina de estados finitos representando o protocolo de comunicação do pipeline de visualização do TerraME. Todos os estados possuem uma transição para o estado de erro que foram omitidas para evitar confusão na figura. O estado erro é atingido sempre que o formato da mensagem do protocolo não é respeitado.. 30

Figura 3.11 – API de criação de observers na linguagem TerraME 31

Figura 3.12 – Diversos usos da função `notify()`. 32

Figura 3.13 – Diversos tipos de observadores da plataforma de modelagem TerraME....................... 34

Figura 4.1 – Somatório de *bytes* transmitidos via *pipeline* avaliando o uso do *blackboard* (BB) e, em seguida, combinado o BB com o *Protocol Buffer* (ProtBuff) como estratégias para redução do número de *bytes* transmitidos. Em (a) os resultados do Teste 1 e em (b) os resultados do Teste 2. 41

Figura 4.2 – Tempo médio de resposta da arquitetura TerraME *Observer*.. 43

Figura 4.3 – Consumo de memória 45

Figura 4.4 – Sobrecarga causada pelo algortimo utilizado para transmitir apenas mudanças do estado interno dos modelos. 47

Figura 4.5 – Tempo de resposta vs. Porcentagem de variação.. 48

LISTA DE SIGLAS E ABREVIATURAS

2D	*Duas dimensões*
3D	*Três dimensões*
ABM:	*Agent-Based Modeling*
API:	*Application Programming Interface*
CPU	*Central Processing Unit*
GUI:	*Graphic User Interface*
GPU	*Graphic Processing Unit*
IDL	*Interface Definition Language*
JVM:	*Java Virtual Machine*
MPI:	*Message Passing Interface*
MVC:	*Model-View-Controller*
XML:	*Extensible Markup Language*

SUMÁRIO

RESUMO iv
LISTA DE ILUSTRAÇÕES E FIGURAS v
LISTA DE SIGLAS E ABREVIATURAS vii
SUMÁRIO viii
1 Introdução 9
1.1 Objetivos 11
1.2 Justificativa 11
2 Fundamentação teórica 13
2.1 *Pipelines* de visualização científica 13
2.2 Plataforma de modelagem ambiental TerraME 14
2.3 Padrões de Projeto de *Software* 17
2.3.1 Padrão de projeto de software Observer 18
2.3.2 Padrão de projeto software BlackBoard 18
3 Metodologia 20
3.1 Requisitos do *pipeline* 21
3.2 Seleção da plataforma de modelagem ambiental para implementação do *pipeline* 21
3.3 Visão geral do projeto do *pipeline* 21
3.4 Arquitetura do *pipeline* 24
3.5 Protocolo de comunicação do *pipeline* 30
3.6 Interface de programação do *pipeline* 31
3.7 Projeto das visualizações de modelos ambientais 34
3.8 Seleção das métricas de desempenho 35
3.9 Definição do ambiente de teste 36
3.10 Planejamento dos experimentos 36
4 Resultados obtidos 40
4.1 Formação de recursos humanos 40
4.2 Publicações 40
4.3 Resultados científicos 40
5 Conclusão 49
Referências 50

1 INTRODUÇÃO

A visualização está presente na história da humanidade há milhares de anos quando informações eram pintadas nas paredes de cavernas. No entanto, somente a partir do século XVII, essa técnica passou a empregar tabelas, diagramas geométricos e outras representações visuais (CHEN; HRDLE; UNWIN, 2008; WARE, 2004). Em 1987, o termo Visualização Científica foi cunhado abrindo um novo campo de pesquisa em que a visualização é aplicada sobre dados científicos (MCCORMICK, 1987).

A visualização tornou-se uma das mais poderosas ferramentas disponíveis para transformar os milhares de dados produzidos diariamente em informações relevantes. Ela permite ao homem sintetizar e analisar grande volume de dados. Desta maneira, ela facilita a compreensão dos fenômenos descritos pelos dados, além da elaboração de hipóteses e da descoberta de conhecimento sobre esses fenômenos. Ela também é aplicada como forma eficaz de comunicação e como ferramenta de apoio em inúmeras áreas: medicina, engenharia, matemática e modelagem ambiental (DEFANTI; BROWN; MCCORMICK, 1989; KELLEHER; WAGENER, 2011; WARE, 2004).

Dentre as áreas de aplicação da visualização científica, a área de modelagem ambiental é de especial interesse para este trabalho. A modelagem ambiental é uma técnica que permite o desenvolvimento de modelos ambientais, ou seja, o desenvolvimento de representações computacionais simplificadas de fenômenos sociais, naturais e suas interações (MULLIGAN, 2004). Dentre estes fenômenos, destacam-se o desmatamento ou o crescimento de uma floresta, a propagação de um incêndio em um parque nacional e a dispersão de uma doença em uma cidade. Como tais fenômenos podem evoluir no tempo e no espaço, os modelos ambientais são utilizados na realização de experimentos simulados que avaliam questões relacionadas tanto à quantidade quanto à localização das mudanças observadas. Desta maneira, os experimentos simulados podem resultar em um grande volume de séries temporais de dados espaciais que precisam ser visualizados e analisados. Algumas vezes, os dados gerados recobrem imensas regiões geográficas em diversas resoluções espaciais e temporais (SCHREINEMACHERS; BERGER, 2011; SPRUGEL et al., 2009; MOREIRA et al., 2009). Nesse sentido, a modelagem e a visualização são vistas como técnicas de apoio à tomada de decisão e à definição de políticas públicas.

Trabalho propôs o desenvolvimento e a avaliação de uma arquitetura de alto desempenho, extensível e flexível para um *pipeline* dedicado à visualização de modelos ambientais. A arquitetura proposta foi implementada na forma de uma tecnologia de código aberto chamada TerraME *Observer* que é disponibilizada gratuitamente junto com a plataforma de modelagem TerraME (endereço *web*: www.terrame.org), para os sistemas operacionais Windows, Linux e Mac.

Este projeto foi conduzido em ciclos de desenvolvimento que produziam versões cada vez mais eficientes da arquitetura de *pipeline* anterior. A cada ciclo, os requisitos que o *pipeline* eram identificados e detalhados. O projeto arquitetural do *pipeline* foi refinado e implementado como uma extensão da plataforma de modelagem ambiental TerraME. Assim, experimentos eram planejados e executados que com objetivo de analisar o desempenho da arquitetura considerando-se as métricas: tempo de resposta observado pelo usuário do *pipeline*, número de *bytes* transmitidos via *pipeline* e consumo de memória pelas estruturas de dados do *pipeline*.

No âmbito do Projeto URBIS Amazônia, esse trabalho contribuiu para (i) formação de recursos humanos: dois alunos tutoreado durante a Iniciação Científica, (ii) publicações: artigo publicado no GeoInfo 2012 e no JIDM 2013 e (iii) resultados científicos: a arquitetura final é capaz de reduzir em 60% a 80% o tempo de resposta observado pelo usuário quando estruturas intermediárias de armazenamento e técnicas de processamento paralelo foram utilizadas de forma combinada na implementação do *pipeline*. Quando a aquisição dos dados a partir dos modelos foi a atividade mais custosa do *pipeline*, as estruturas intermediárias de armazenamento tiveram um papel central no ganho de desempenho. Já quando as atividades de mapeamento visual, *rendering* e exibição foram mais custosas, as técnicas de processamento paralelo trouxeram mais vantagens. O acréscimo máximo no consumo de memória observado durante os experimentos foi inferior a 7%. O uso de estruturas de armazenamento intermediário reduziu o número de *bytes* transmitidos via *pipeline* em aproximadamente 60% quando mais de uma visualização observava os dados de um mesmo modelo.

1.1 Objetivos

Este trabalho teve como objetivo desenvolver e a avaliar uma arquitetura de alto desempenho para a visualização de simulações ambientais. A arquitetura desenvolvida atendeu os requisitos de extensibilidade, adição de novos tipos de visualizações, e de flexibilidade, personalização na forma de observar componentes de modelos definidos pelo usuário e também permitiu o monitoramento em tempo de simulação da dinâmica das variáveis presentes no modelo.

Os objetivos específicos deste incluíram: (i) implementar e disponibilizar a arquitetura proposta na forma de uma tecnologia de código aberto e gratuitamente distribuída junto com uma plataforma de modelagem ambiental, (ii) planejar, executar, analisar e documentar experimentos que permitissem avaliar o desempenho da arquitetura proposta.

1.2 Justificativa

Simulações ambientais são capazes de produzir volumes massivos de dados que precisam ser visualizados, analisados e interpretados. Por esta razão, *pipelines* de visualização científica destinados ao monitoramento e à análise de simulações ambientais podem promover o desenvolvimento de modelos ambientais à medida que facilitam a interpretação dos dados produzidos pelas simulações e, portanto, também facilitam a identificação de falhas conceituais e de implementação em tais modelos. Neste contexto, a visualização científica pode ser vista como um importante instrumento de apoio à tomada de decisão e à definição de políticas públicas.

No Brasil, questões de interesse público são respondidas por meio de modelos ambientais implementados na plataforma TerraME. Estes modelos produzem prognóstico sobre evolução espaço-temporal das interações sociedade-natureza em diversos domínios: desflorestamento na Floresta Amazônica brasileira (AGUIAR; CÁMARA; ESCADA, 2007; CÂMARA et al., 2005; MOREIRA et al., 2009), emissão de gases de efeito estufa (AGUIAR et al., 2012), produção de grãos (MARTORANO et al., 2009), propagação de incêndio em parques nacionais (ALMEIDA; MACAU, 2011; ALMEIDA et al., 2008), deslizamento de terra (LOPES; NAMIKAWA; REIS, 2011; REIS; CORDEIRO; LOPES, 2011), relações e interações socioeconômicas na Amazônia contemporânea (AMARAL et al., 2013; CARDOSO; VENTURA NETO, 2013; LIMA; SIMÕES; MONTEMÓR, 2013) e dispersão da dengue (LANA et al., 2010, 2011). Contudo, até o inicio deste projeto a plataforma

TerraME não oferece serviços de monitoramento em tempo-real das simulações que é capaz de produzir. Por estas razões, um *pipeline* de visualização científica destinada ao monitoramento e análise de simulações ambientais encontraria respaldo em diversas comunidades científicas ocupadas em compreender e produzir prognósticos a cerca das interações entre o homem e o ambiente.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta os conceitos básicos necessários ao entendimento do trabalho e descreve os principais trabalhos correlatos nos temas *pipelines* de visualização científica. Os métodos e ferramentas de visualização científica apoiam tanto a análise dos resultados produzidos por um modelo quanto ao desenvolvimento do próprio modelo, à medida que ajudam na identificação de falhas conceituais e de implementação do mesmo.

2.1 *Pipelines* de visualização científica

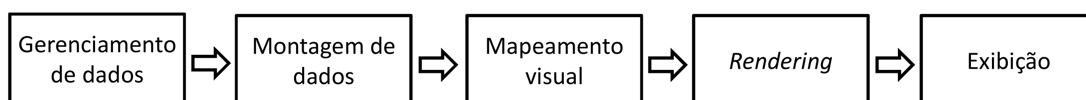


Figura 2.1 – Estágios do processo de visualização. Adaptado de (WOOD et al., 2005).

Uma rede de fluxo de dados ou *pipeline* é uma sequência de estágios interconectados em que cada estágio realiza uma transformação sobre os dados de entrada e os fornece ao próximo estágio. Nos *pipelines de* visualização científica estas transformações visam produzir uma representação pictural dos dados de entrada. Formalmente, o termo *pipeline de visualização* é grafo direcionado onde os nós representam módulos de processamento e as arestas (conexões) representam alguma dependência de dados (MORELAND, 2013; VO et al., 2010; WOOD et al., 2005), como mostrado na Figura 2.1.

O termo módulo é recentemente generalizado por Moreland (2013) como "uma unidade funcional de rede" enquanto Upson (1989) o caracterizou de acordo com suas conexões de entrada e saída:

- Módulos de origem: possuem apenas conexões de saída e representam as fontes de dados;
- Módulos transformadores: têm inúmeras conexões tanto de entrada quanto de saída de dados e representam qualquer tipo de transformação nos dados (por exemplo: filtros, interpolação, *rendering*, etc.);
- Módulos terminais: possuem apenas conexões de entrada e representam o resultado final do *pipeline* (por exemplo: a tela do monitor e impressora)

As conexões permitem que artefatos de um módulo fluam para módulos seguintes através das portas de entrada e saída de dados.

Wood et al (2005) identificaram cada estágio de um *pipeline* de visualização (Figura 2.1), como sendo:

- Gerenciamento de dados: estágio responsável pelo acesso e aquisição dos dados brutos;
- Montagem de dados: fase em que algum pré-processamento ou transformação é aplicada sobre dado (por exemplo: filtragem, agregação, composição, etc.) gerando dados com propósitos de visualização;
- Mapeamento visual: converte e transforma os dados recebidos em alguma representação visual, atribuindo características gráficas como coordenadas e cores;
- *Rendering*: a partir do ponto de vista do usuário do sistema, este estágio produz imagens bidimensionais dos dados recebidos;
- Exibição: apresenta as imagens geradas pelo *rendering* em algum dispositivo de saída (por exemplo: a tela do monitor, impressora ou óculos de realidade virtual).

2.2 Plataforma de modelagem ambiental TerraME

O *Terra Modeling Environment* – TerraME[1] é destinado ao desenvolvimento de modelos em múltiplas escalas e espacialmente-explícitos (CARNEIRO et al., 2013). Além de integrar o conjunto de plataformas ABM, ele também permite o desenvolvimento de modelos baseados em outros paradigmas, por exemplo, teoria geral de sistemas, teoria de autômatos celulares, especificação de sistemas por eventos discretos (DEVS), entre outros (VON NEUMANN, 1966; WOOLDRIDGE; JENNINGS, 1995; ZEIGLER; PRAEHOFER; KIM, 2000).

O ambiente TerraME foi projetado para atender usuários de diversas formações que, em grande parte, são especialistas no domínio do problema. Para atender esses usuários, a plataforma dispõe de uma linguagem de alto-nível chamada TerraME, que estende a linguagem Lua (IERUSALIMSCHY; FIGUEIREDO; FILHO, 1996; IERUSALIMSCHY, 2006) adicionando novos tipos especialmente desenvolvidos para a modelagem ambiental. Eles permitem descrever características comportamentais, temporais e espaciais dos fenômenos em estudo. Características espaciais são expressas pelos tipos *Cell*, *CellularSpace*, e *Neighborhood* e representam propriedades inerentes as localizações e relações geográficas de proximidade. Características comportamentais permitem representar como atores e processos alteram propriedades do espaço. Elas são modeladas pelos tipos *Agent, Automaton*

---
[1] www.terrame.org

e *Trajectory* que representam, respectivamente, indivíduos, campos dinâmicos e formas de percorrer o espaço. Os tipos *Timer* e *Event* expressam características temporais e representam o tempo da simulação. Modelos em TerraME também podem ser descritos por meio da linguagem C++. Esta abordagem é indicada para usuários com sólidos conhecimentos de programação e que desejam alto desempenho.

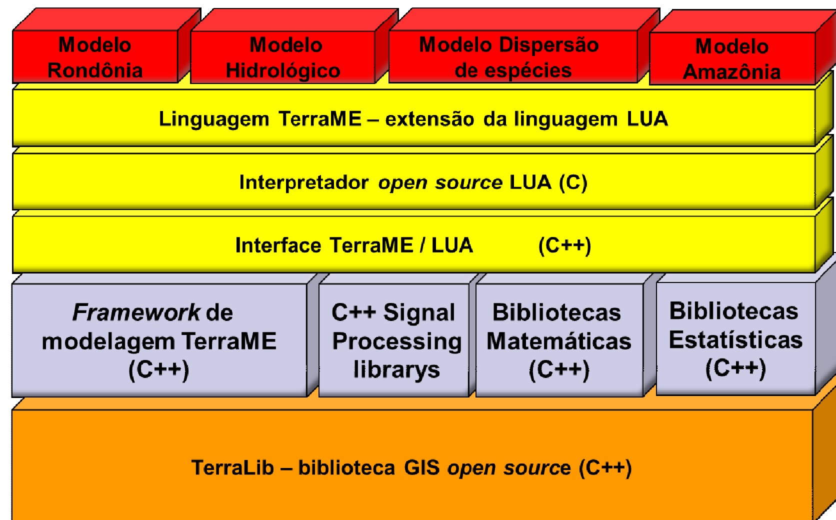


Figura 2.2 – Arquitetura da plataforma de modelagem TerraME. Adaptado de (CARNEIRO et al., 2013).

A arquitetura da plataforma está organizada em camadas (Figura 2.2), na qual camadas superiores utilizam os serviços fornecidos pelas camadas inferiores. Na primeira camada, a biblioteca TerraLib provê serviços de leitura e escrita de dados em banco de dados geográficos (CÂMARA et al., 2008). O núcleo do ambiente TerraME, segunda camada, oferece serviços de modelagem, simulação, calibração e validação. A terceira camada fornece a interface entre TerraME e Lua onde os tipos de objetos e demais serviços estão registrados no ambiente de execução (interpretador ou máquina virtual) Lua. Na quarta camada, o interpretador Lua provê as análises sintática e semântica da linguagem, além de ser ambiente de execução. Na última camada, encontram-se os modelos dos usuários da plataforma TerraME escritos na linguagem de alto nível.

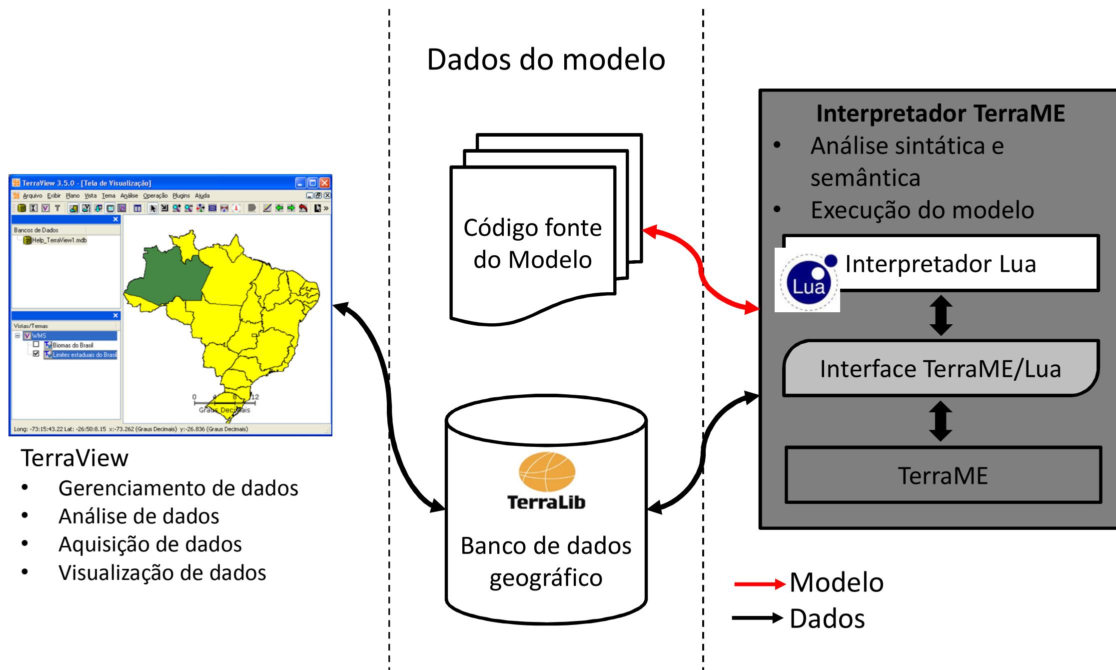


Figura 2.3 – Ambiente de execução do TerraME

O desenvolvimento de modelos pode feito em qualquer editor de texto. O arquivo texto contendo as regras do modelo, isto é, o código fonte do modelo, é passado como parâmetro para o simulador que o interpreta e executa a simulação (Figura 2.3). O resultado gerado pode ser visualizado no ambiente TerraView[2] após o término da simulação. O TerraView é um sistema de informação geográfica (SIG) que integra o pacote de ferramentas TerraLib. Ele permite criar, gerenciar, analisar e visualizar bancos de dados geográficos gerados por meio dessa biblioteca.

Até o inicio deste trabalho, a plataforma TerraME não dispunha de serviços para monitoramento e visualização dos modelos ambientais durante as simulações. Assim, ele foi escolhido porque não possuía visualizações em tempo real, ser uma plataforma livre e de código aberto, devido a sua flexibilidade em permitir o desenvolvimento de modelos ambientais por meio da combinação de vários paradigmas de modelagem e por sua capacidade em representar e simular modelos que consideram múltiplas escalas de um mesmo fenômeno. Estas propriedades são de vital importância para muitas questões científicas e políticas enfrentadas pelo Brasil. Nos modelos de mudança de uso e cobertura do solo (LUCC - *land use and cover change*) utilizados para responder questões acerca do desflorestamento na região da Floresta Amazônica brasileira, o processo de desflorestamento depende de

[2] http://www.dpi.inpe.br/terraview/

diferentes fatores em diferentes escalas (níveis de detalhamento). Na escala global, ele pode ser influenciado pelo preço da carne ou da soja. Na escala regional, ele depende da proximidade de meios de escoamentos, como, estradas e mercados de consumo. Na escala local, depende das condições financeiras do dono da terra: pequeno, médio e grande produtor (AGUIAR; CÁMARA; ESCADA, 2007; MOREIRA et al., 2009). O sistema uso e cobertura do solo influencia e é influenciado diversos outros sistemas socioeconômicos e naturais que Carneiro et. al (2013) não acreditam que um único paradigma de modelagem seja suficiente para produzir modelos realistas e de fácil evolução por equipes multidisciplinares.

Além destas questões, a plataforma TerraME é de código aberto, extensível, permite a construção de novos tipos em C++ e o posterior registro destes tipos na linguagem de alto nível derivada de Lua. Assim, os tipo *observer* e *subject* poderiam ser facilmente adicionados à linguagem TerraME. Até o início deste projeto, o TerraME não possuía mecanismos para visualização das simulações em execução. Os resultados eram observados na forma de mapas ou imagens somente após persistidos em um banco de dados geográfico. Isto dificultava a identificação de falhas conceituais ou de implementação dos modelos durante o processo de desenvolvimento.

2.3 Padrões de Projeto de *Software*

Vários desafios computacionais envolvidos no projeto, construção e teste de um *pipeline* de visualização científica para simulações ambientais são, na verdade, problemas recorrentes em diversos projetos de sistemas de computação. Por isto, se faz necessária revisão dos padrões de projeto de software que podem apoiar o desenvolvimento deste projeto. Um padrão de projeto de *software* é uma sequência de ações recorrentes e abstratas aplicadas à solução de um problema computacional dentro de um contexto específico (BUSCHMANN et al., 1996; GAMMA et al., 1995). Os padrões são conceituados por Gamma et al. (1995, p. 20) como: “(...) descrição de classes e objetos comunicantes que são personalizados para resolver um problema geral de projeto em um contexto particular.”

2.3.1 PADRÃO DE PROJETO DE SOFTWARE OBSERVER

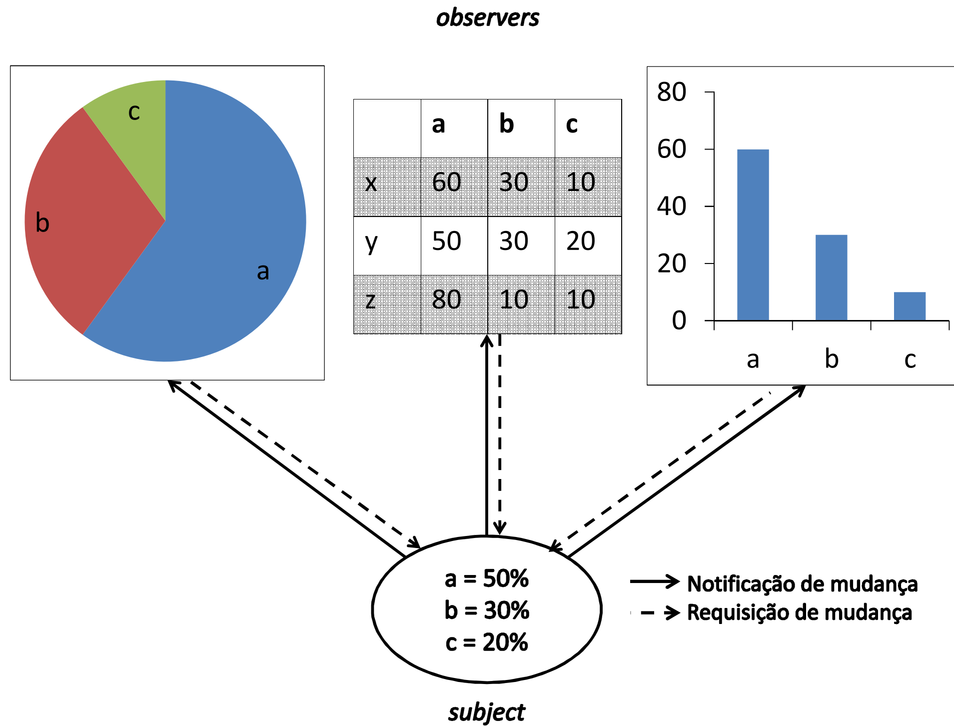


Figura 2.4 – Padrão de projeto de *software Observer*. Adaptado de (GAMMA et al., 1995)

O padrão de projeto de *software Observer* é um padrão comportamental também conhecido como *Model-View-Controller* (MVC) ou *Publish-subscribe*. Ele permite relacionar dois objetos mantendo-os fracamente acoplados de maneira que possam ser usados separadamente, maximizando a reusabilidade do código (GAMMA et al., 1995).

Os atores principais desse padrão são os objetos *subject* e *observer*. O *subject* tem seu estado interno monitorado e, quando alterado, notifica o *observer*. Por sua vez, o *observer* recupera automaticamente esse estado, se atualiza e o apresenta de forma apropriada (Figura 2.4). De maneira simplificada, os observadores são interfaces gráficas do usuário (GUI's) que apresentam visões sobre o dado contido no *subject.* Um *subject* pode não ter nenhum observador ou estar associado a vários. Um *observer* deve estar associado a apenas um *subject*.

2.3.2 PADRÃO DE PROJETO SOFTWARE BLACKBOARD

O sistema *BlackBoard* foi utilizado pela primeira vez na área de Inteligência Artificial na década de 1960 por Allan Newell (NII, 1986c), sendo utilizado por subsistemas

especializados como uma área comum de integração de conhecimento para a construção parcial ou aproximada de soluções de problemas não-determinísticos (BUSCHMANN et al., 1996; NII, 1986a, 1986b). Newell vislumbrou uma área compartilhada, um quadro negro, onde informações ficam armazenadas de maneira que especialistas possam ler, julgar, adicionar e excluir essas informações em busca da solução do problema em questão. A cada instante, um controlador seleciona um especialista de acordo com o estado atual do problema e coordena seu acesso ao quadro. Esse especialista, através de suas experiências, tenta avançar em direção à solução e qualquer informação útil gerada é disponibilizada no quadro negro. Desta forma, todos os envolvidos têm acesso às subsoluções, sejam elas parciais ou aproximadas (BUSCHMANN et al., 1996; CORKILL, 1991; NII, 1986a, 1986b).

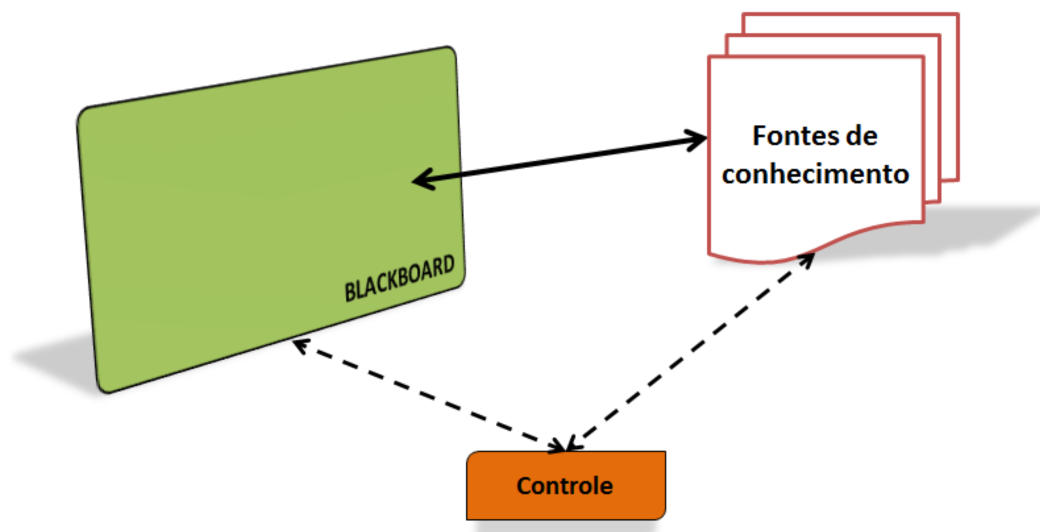


Figura 2.5 – Diversos visualizadores compartilham as informações mantidas no *BlackBoard*

Um *blackboard* poderia ser utilizado como uma memória *cache* intermediária a dois estágios de um *pipeline* de visualização científica. Nesta arquitetura, o módulo de controle implementaria diferentes políticas de atualização da cache. *BlackBoard* é um padrão arquitetural que determina uma disposição organizacional básica, constituído como uma estrutura de armazenamento central, diversas fontes de conhecimento e um módulo de controle, como ilustra a Figura 2.5 (BUSCHMANN et al., 1996).

Buschmann (1996) identifica o *Repositório* como uma generalização do *BlackBoard* por não possuir um módulo de controle interno. Nessa variação, o controle pode ser feito por um programa externo ou pelo usuário, como ocorre em aplicativos baseados em bancos de dados e ambientes colaborativos multiusuários.

3 METODOLOGIA

Esta seção apresenta a metodologia utilizada no desenvolvimento deste projeto. Para que fosse possível projetar, construir e avaliar um *pipeline* de alto desempenho para visualização de simulações ambientais, este projeto se dividiu em várias fases que foram executadas ciclicamente. A cada ciclo, uma versão melhorada do *pipeline* foi desenvolvida e avaliada por meio de experimentos que quantificassem seu desempenho na visualização em tempo real das simulações. Durante este processo, também foram privilegiados requisitos como a extensibilidade do *pipeline* por meio da fácil adição de novas formas de síntese (transformação) e visualização de dados. Além de permitir que as implementações e instâncias das visualizações fossem fracamente acopladas aos modelos ambientais.

Desta maneira, pode-se dizer que este projeto tem um forte caráter experimental. À medida que uma versão do *pipeline* era avaliada em experimentos de análise de desempenho e planejamento de capacidade, os gargalos do sistema eram identificados. Uma vez identificados, novos métodos e técnicas de computacionais eram utilizados para dirimir estes gargalos.

Na fase de concepção, buscou-se identificar e documentar os requisitos funcionais e não funcionais desse *pipeline*. Na fase de projeto, foram utilizados os conhecimentos sobre algoritmos de indexação e estruturas de dados, projeto de protocolos de comunicação, projeto de algoritmos de compactação, projeto de políticas de atualização de *cache* e projeto de algoritmos paralelos para melhorar gradativamente o *pipeline* de visualização desenvolvido. Na fase de construção, o *pipeline* foi implementado como uma extensão da plataforma de modelagem ambiental TerraME e sua implementação foi verificada por meio de testes unitários e funcionais. Na fase de teste, diversos experimentos para avaliação de desempenho e planejamento de capacidade (JAIN, 1991) foram planejados, realizados, analisados e documentados.

Ao longo desta seção, descrevemos o *pipeline* desenvolvido, discutimos seus requisitos, justificamos as decisões de projeto tomadas e o planejamento dos experimentos, realizados.

3.1 Requisitos do *pipeline*

Alguns requisitos foram considerados essenciais para a construção de um *pipeline* de visualização de alto desempenho e são descritos a seguir:

- Requisitos funcionais:
  - Apresentar graficamente a dinâmica das variáveis de estado contínuo e discreto
  - Fornecer visualizações para as dimensões temporal, espacial e comportamental de um modelo ambiental.
  - Exibir graficamente a co-evolução de variáveis de estado contínuo e discreto de forma que os padrões de mudança/comportamento possam ser identificados e compreendidos mais facilmente.

- Requisitos não funcionais:
  - Mudanças na variável de estado devem ser apresentadas em tempo real e devem impactar o mínimo possível no desempenho de simulação.
  - Deve ser extensível para que novas visualizações possam ser facilmente desenvolvidas pelo usuário.
  - Modelo contendo visualizações devem continuar podendo ser simulados em versões antigas das plataformas de simulação que intencionarem adotar o projeto do *pipeline* desenvolvido neste trabalho. Desta maneira, o *pipeline* deve ser fracamente acoplado aos modelos de forma a garantir a portabilidade desses modelos.

3.2 Seleção da plataforma de modelagem ambiental para implementação do *pipeline*

Dentre as plataformas de modelagem avaliadas na Subseção 2.2,

3.3 Visão geral do projeto do *pipeline*

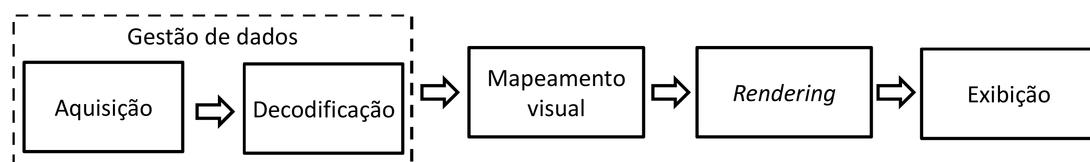


Figura 3.1 – Visão geral do *pipeline* de visualização do TerraME. Adaptado de Wood et al. (2005)

O *pipeline* de visualização construído neste trabalho foi baseado naquele apresentado em Wood et al. (2005) e apresentado na Subseção 2.1. A Figura 3.1 demonstra uma visão geral deste *pipeline*, no qual, o módulo de gestão de dados é dividido em Aquisição e Decodificação. Os próximos módulos, mapeamento visual, *rendering* e exibição, são os mesmo apresentados na Subseção 2.1. As responsabilidades de cada módulos são:

- Módulo de aquisição: É responsável pelo acesso à fonte de dados, por exemplo, arquivo em disco, espaço de memória da linguagem de alto nível ou acesso ao banco de dados, e codificação desses dados seguindo o protocolo de comunicação (apresentado na Subseção 3.5). Essa codificação é necessária para que o requisito de fraco acoplamento seja atendido.
- Módulo de decodificação: Decodifica os dados recebidos de acordo com o protocolo de comunicação e prepara-os para o próximo módulo.
- Módulo de mapeamento visual: Gera uma representação visual dos dados, por exemplo, atribuição de cores ou formas aos elementos de dados.
- Módulo de *rendering*: Constrói uma imagem bidimensional a partir dos artefatos recebidos do módulo anterior.
- Módulo de exibição: Apresenta as imagens renderizadas em algum dispositivo de saída, por exemplo, impressora, telas, etc.

A atual versão do *pipeline* de visualização não inclui o módulo de Montagem cuja complexidade escapa ao escopo deste trabalho e é uma excelente fonte de contribuições futuras. Quando inserido no *pipeline*, deveria ser inserido entre os módulos de decodificação e mapeamento visual, conforme Figura 2.1. Atualmente, os artefatos do módulo de decodificação fluem diretamente para o módulo de mapeamento visual.

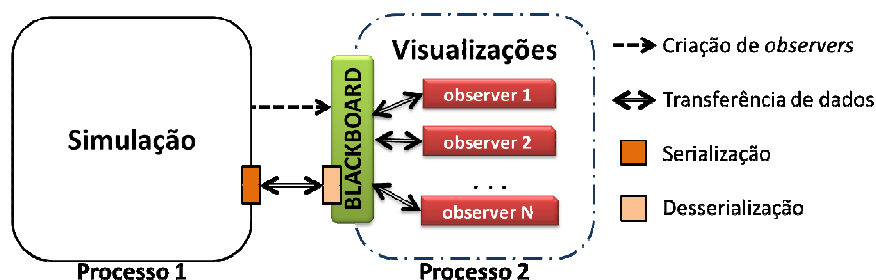


Figura 3.2 – Simulação e visualizações podem residir em espaços de memória de processos distintos ou em camadas de software distintas. No TerraME, as variaveis de estado do modelo estão na pilha do interpretador Lua e as visualizações estão no *framework* C++.

O módulo de Aquisição faz o acesso ao dado bruto, reúne e serializa os dados seguindo o protocolo descrito adiante na Subseção 3.5. O uso de um protocolo formalmente definido e bem documentado desacopla os outros módulos do *pipeline* das estruturas de dados utilizadas para representar o modelo. Para obter os dados do modelo, eles precisam apenas conhecer o protocolo na qual os dados serão transmitidos. A etapa de Decodificação desserializa o dado e o repassa para os outros módulos do *pipeline*. Desta maneira, os módulos de aquisição e decodificação implementam as duas entidades envolvidas em uma comunicação, aquisição é o transmissor e decodificação é o receptor.

Algumas vezes o estado interno de um modelo deve ser adquirido a partir de um espaço de endereçamento diferente daquele onde ocorrem as visualizações. Nestas situações, os dados precisam ser adquiridos a partir do espaço de endereçamento do modelo, transmitido por um meio de um protocolo e, finalmente, armazenados em outro espaço de endereçamento acessível pelos módulos de mapeamento visual, *rendering* e exibição. Estes espaços de endereçamento podem ser relativos a processos ou camada de *software* diferentes. Conforme ilustra a Figura 3.2, o modelo pode residir em um processo enquanto que as visualizações são construídas por outro. Estes processos podem inclusive executar em equipamentos diferentes. Não é raro encontrar ambientes nos quais os modelos executem em *clusters* de processamento (supercomputadores) dedicados às simulações e as visualizações executem em estações gráficas especializadas na visualização científica de grandes volumes de dados. No caso do TerraME, os dados do modelo precisam ser adquiridos a partir da pilha de execução do interpretador Lua, localizado na terceira camada da arquitetura TerraME, e precisam ser mapeados, renderizados e exibidos por interfaces gráficas implementadas em C++ e disponíveis no *framework* de modelagem localizado na segunda camada da arquitetura TerraME (Figura 2.2).

3.4 Arquitetura do *pipeline*

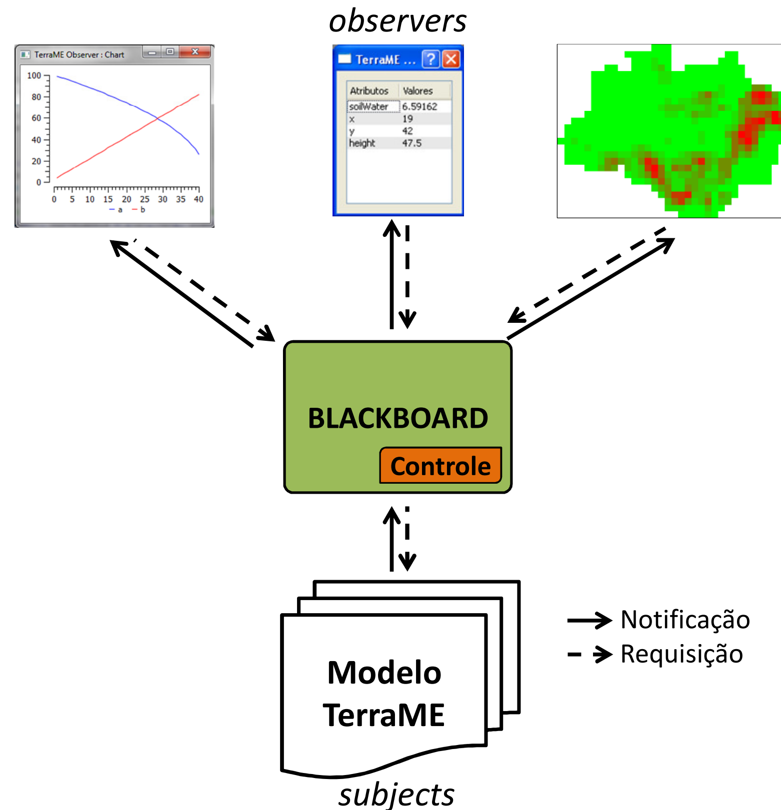


Figura 3.3 – Estruturação do padrão *Observer* e do padrão *BlackBoard* no pipeline do TerraME

A arquitetura do pipeline segue o padrão de projeto de *software Observer* combinada com o padrão *BlackBoard* descritos na Subseção 2.3.2. A maior parte dos tipos de objeto TerraME tornara-se *subjects* (por exemplo: *Cell*, *CellularSpace*, *Agent*, *Automaton*, *Scheduler*, entre outros) e uma variedade interfaces gráficas que exibem visualizações de instâncias destes tipos, doravante chamadas de *observers*, foram desenvolvidas (por exemplo, gráficos, tabelas, imagens 2D, etc.). Desta maneira, durante uma simulação os estados internos dos *subjects* podem ser visualizados de várias formas e simultaneamente, por vários *observers* que executam em paralelo como esboçado na Figura 3.2 e na Figura 3.3. Para situações em que os observadores predefinidos não são adequados, a arquitetura de *pipeline* implementada oferece uma *Application Programming Interface* (API) para a criação de novos tipos de *observers*.

O padrão *BlackBoard* é usado como memória *cache* temporária compartilhada entre os módulos de gestão de dados e mapeamento visual do *pipeline*. Quando armazenados no *blackboard*, os dados podem ser visualizados por diferentes observadores sem que os módulos de *aquisição e decodificação* de dados sejam executados novamente. Os

experimentos realizados nesta projeto demonstraram que estes dois módulos são os mais lentos do *pipeline*.

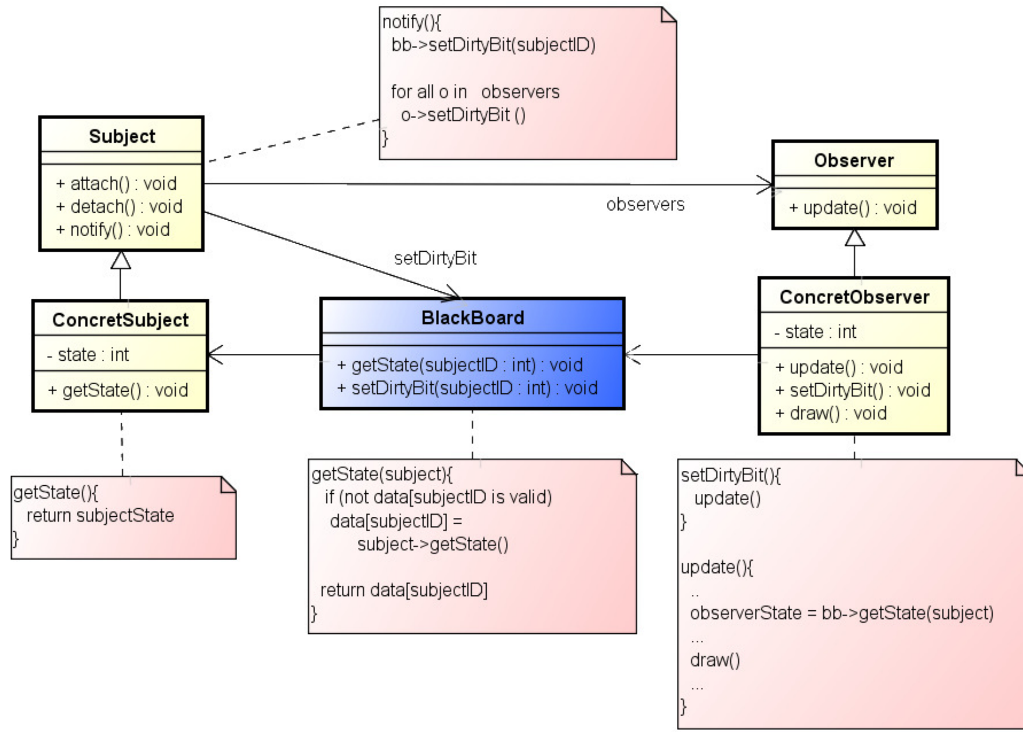


Figura 3.4 – Diagrama de classe das principais entidades envolvidas na comunicação entre modelo e visualização – Integração entre os padrões de projeto *Observer* e *Blackboard*

Os observadores substituem as fontes de conhecimento do padrão *Blackboard.* Eles escrevem no *blackboard* à medida que requisitam a atualização dos dados sobre os *subjects*. Logo, o *pipeline* possui uma política de atualização por demanda. Isto permite o uso de um *dirty bit* para indicar que algum dado na *cache* foi invalidado e precisa ser atualizado.

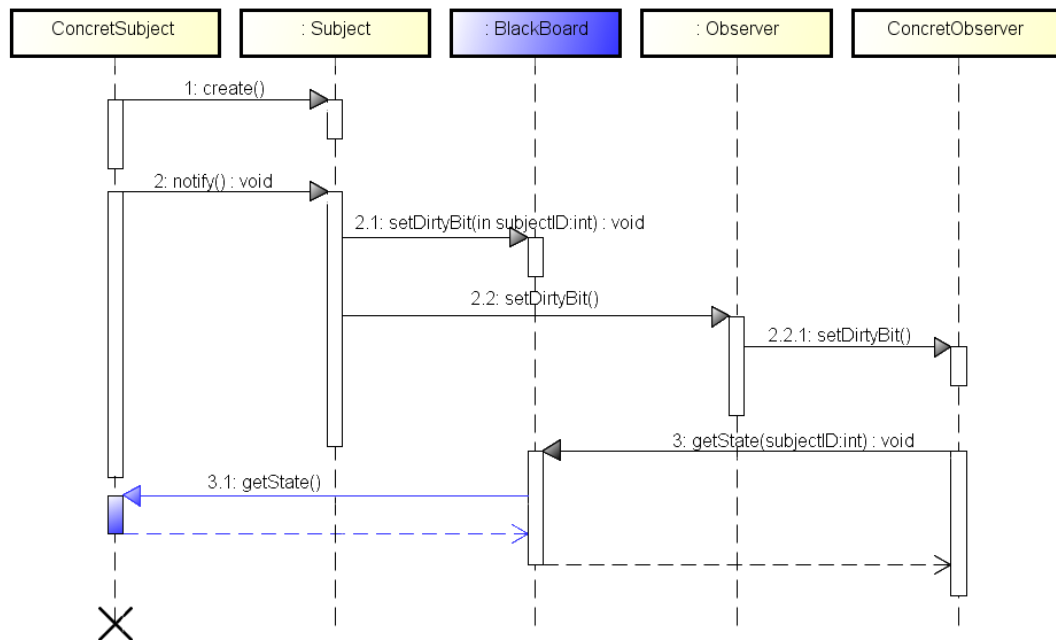


Figura 3.5 – Diagrama de sequência – Interações entre os padrões de projeto *Observer e Blackboard*

A Figura 3.4 e a Figura 3.5 mostram em detalhes a integração entre os padrões de projeto de *software Observer* e *BlackBoard* para implementar a comunicação (fluxo de dados) entre modelo e visualizações. Conforme diagrama de sequência apresentado na Figura 3.5, o processo de visualização é iniciado sob demanda quando a requisição de atualização do *pipeline,* `notify()`, inserida no código fonte do modelo é invocada. Ela sinaliza que ocorreu alguma alteração no estado interno do *subject* monitorado e, portanto, todos os observadores a ele ligados devem ser atualizados. Estas notificações marcam os dados armazenados no *blackboard* como inconsistentes ou "sujos" e também invalidam as visualizações. Quando o primeiro observador do *subject* notificado ganha acesso a uma CPU e iniciar sua atualização, ele invoca o método `getState()` desse *subject* para recuperar seu estado interno. Este método faz a aquisição e a decodificação dos dados, armazenado-os no *blackboard*. Ele também atualiza o *dirty bit* para "true" indicando que os dados armazenados são válidos (consistentes, atualizados ou limpos). Quando os próximos observadores deste *subject* ascenderem a uma CPU, os dados no *blackboard* já estarão decodificados e atualizados. Logo, o método `getState()` do *subject* não será invocado. Desta maneira, os módulos de gestão de dados do *pipeline* não precisam ser executados, melhorando o desempenho do fluxo de dados.

Além do ganho de desempenho trazido pelo uso do *dirty bit*, a decisão de implementar uma política de atualização sob demanda também pode ser justificada pelo fato que, deste modo,

as visualizações sempre apresentarão estados das computações de uma simulação que são de interesse do usuário e por ele selecionados. A seleção do estado a ser visualizado é importante durante a verificação de um modelo, quando se faz necessário determinar o efeito de qual computação se deseja investigar.

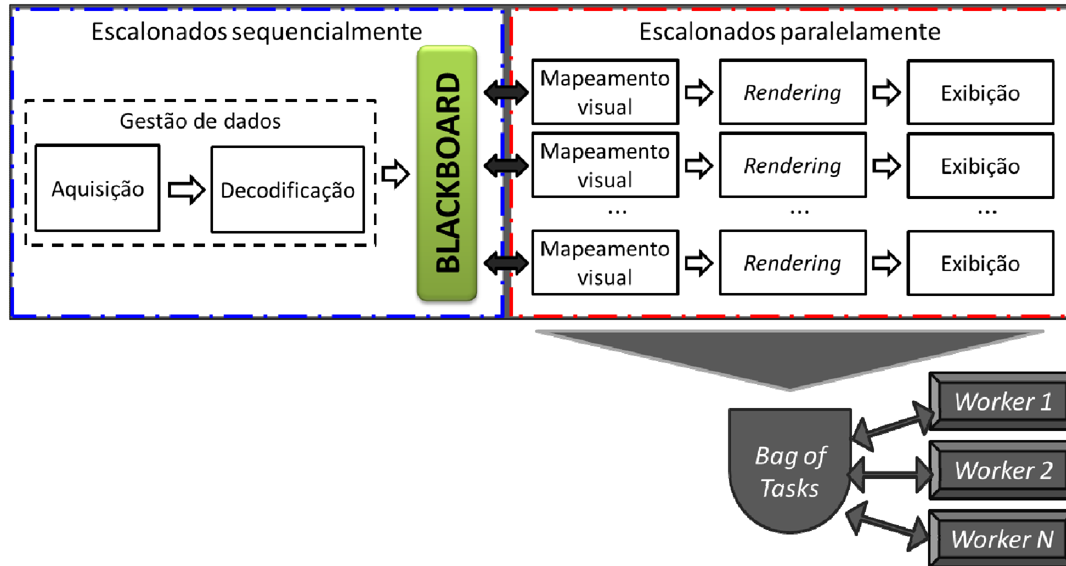


Figura 3.6 – Detalhamento do escalonamento dos módulos

No intuito de melhorar o desempenho da arquitetura proposta, mecanismos para permitir o paralelismo de *pipeline* foram incorporados. Os módulos de aquisição e decodificação são escalonados de maneira sequencial enquanto que a sequência posterior de módulos (mapeamento visual, *rendering* e exibição) são escalonados em paralelo, conforme ilustra a Figura 3.6. Assim, a atualização do *pipeline* acontece de forma assíncrona. Chamadas à função `notify()` apenas atualizam o *blackboard.* Enquanto que, os *pipelines* paralelos simultaneamente atualizam as visualizações.

O mecanismo de *Bag of Tasks* (BEAUMONT et al., 2008; BENOIT et al., 2010) foi utilizado para paralelizar as transformações realizadas pelos módulos do *pipeline* e para balancear a carga de trabalho entre os diversos processadores disponíveis. Desta forma, à medida que os módulos necessitam transformar um dado, eles inserem esta transformação em uma fila de tarefas esperando para serem executadas. Esta fila recebe o nome de *bag*. As tarefas enfileiradas serão executadas por *threads* denominadas *workers* que compartilham o acesso à *bag* e, continuamente, consomem tarefas, executando-as e repassando o resultado aos módulos que as demandaram. Logo, as transformações são executadas na ordem em que são

solicitadas. O número de *workers* instanciados pode ser definido pelo usuário ou é igual ao número de processadores disponíveis.

Neste novo contexto, as atualizações do *blackboard* a partir dos *subjects* e a leitura de dados pelos *pipelines* paralelos ocorrem de forma concorrente. No entanto, um *pipeline* paralelo não deve obter dados que estão em atualização. Portanto, os acessos ao *blackboard* precisam ser sincronizados para que as visualizações sejam consistentes. Neste trabalho, os elementos de dados de um determinado *subject* é bloqueado no início da sua atualização e permanece neste estado até que atualização seja completamente finalizada. Enquanto um dado estiver bloqueado, qualquer observador que o solicite é retirado do processador e colocado em uma fila de espera pelo dado solicitado. Assim que o dado é desbloqueado, todos os observadores esperando por ele são reativados.

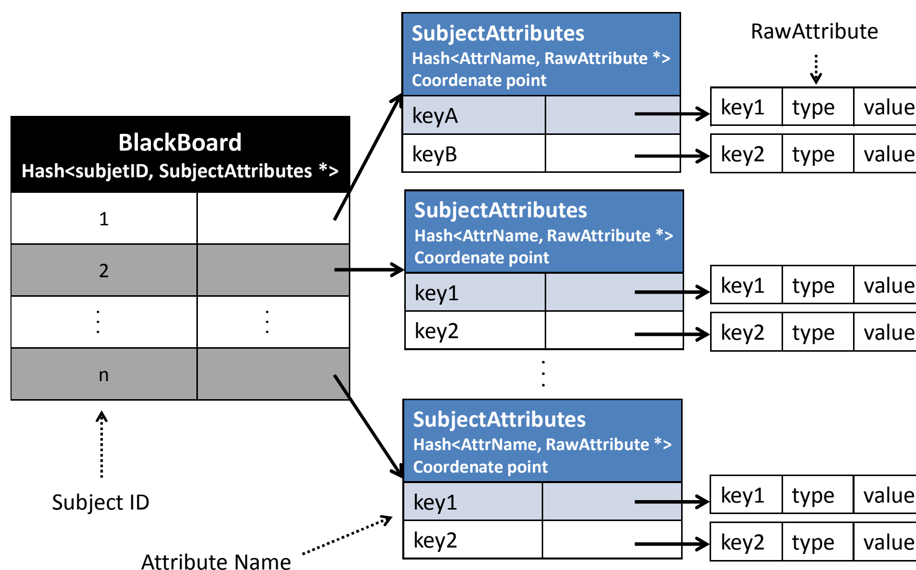


Figura 3.7 – Detalhamento da estrutura de dados que implementa o *BlackBoard*

A estrutura de armazenamento do *blackboard* é implementada como tabelas *hash* que mapeia índice (identificador único) em um elemento de dado intermediário, ou seja, um objeto `SubjectAttributes`. Este objeto representa uma coleção de valores de atributos (objeto `RawAttribute`) que descrevem um *subject* (Figura 3.7). Essa organização faz-se necessária porque há uma cardinalidade 1:n entre *SubjectAttributes* e *RawAttribute* descrita no diagrama de classe (Figura 3.8). A Figura 3.8 apresenta a API de todas as classes envolvidas na implementação do *blackboard*.

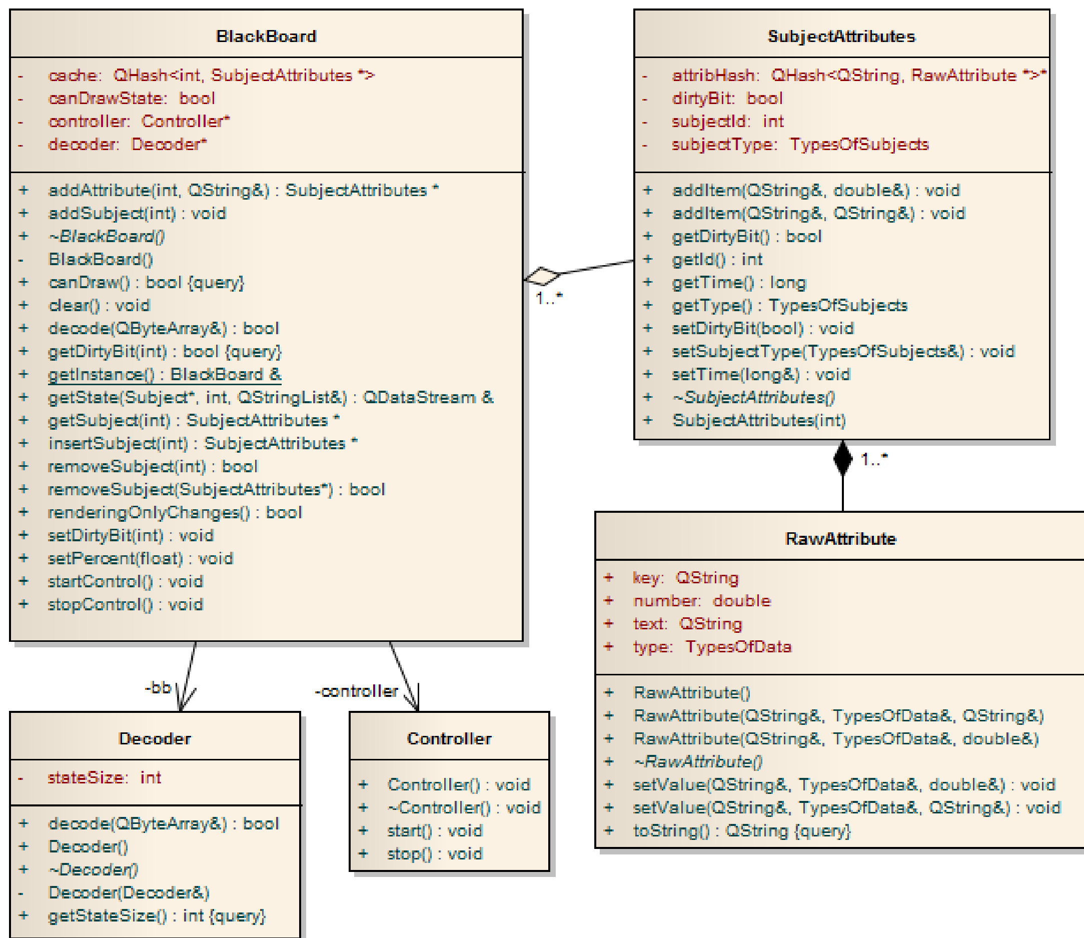


Figura 3.8 – Diagrama de classe do *BlackBoard*

Um ultima melhoria foi incorporada à arquitetura de forma a diminuir o volume de dados transmitido através do *pipeline*. Dado um *subject*, que somente seus dados que sofreram alguma mudança desde a última atualização do *blackboard* são novamente processados pelos módulos de aquisição e decodificação. Por exemplo, se apenas 10% das células de um espaço celular sofreram mudanças desde a última vez que o espaço celular foi visualizado, então somente estas poucas células serão adquiridas e decodificadas. Se nos próximas atualizações os 90% de células inalteradas permanecerem neste estado, então o *controller* do padrão *Blackboard* irá liberar o espaço ocupados por elas na *cache.*

Quando o último observador instanciado é atualizado, o *controller* do padrão *Blackboard* avalia o espaço usado pela *cache* e libera os dados considerados desnecessários. Na primeira vez que a primitiva *notify*( ) é invocada pelo modelo, o *controller* é interrompido. A cada execução, o *controller* avalia apenas uma região da *cache (10%)* para evitar sobrecarga. Desta maneira, o *controller* executa reduz seu impacto sobre o tempo de simulação. Porém, se o

tempo de simulação entre duas chamadas *notify()* for menor que o tempo para atualizar todos os observadores, então o *controller* jamais será iniciado. Avaliar diferentes políticas para acionamento e funcionamento do *controller* é um dos próximos passos a serem seguidos nesta projeto.

3.5 Protocolo de comunicação do *pipeline*

```
<subject>::=<subject identifier><subject type><number of attributes>
<number of internal subjects>[*<attribute>][*<subject>]
<attribute>::=<attribute name><attribute type><attribute value>
```

Figura 3.9 – Formato de datagrama

Os *subjects* e os *observers* são fracamente acoplados de forma que podem ser implementados de maneira independente. Isto é, *observers* podem ser construídos sem o conhecimento de como os *subjects* são implementados. Para isto, toda a comunicação entre estas entidades seguem um protocolo de serialização desenvolvido neste trabalho de acordo com as necessidades do domínio modelagem ambiental. Este protocolo é definido como um conjunto de regras para comunicação e um formato de mensagem bem definidos que regulam seu uso (POPOVIC, 2006).

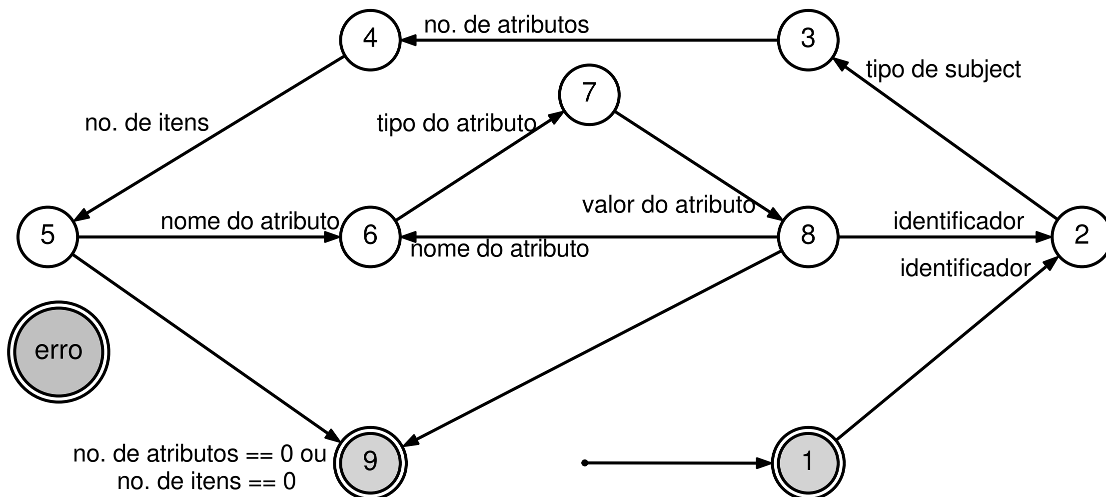


Figura 3.10 – Máquina de estados finitos representando o protocolo de comunicação do pipeline de visualização do TerraME. Todos os estados possuem uma transição para o estado de erro que foram omitidas para evitar confusão na figura. O estado erro é atingido sempre que o formato da mensagem do protocolo não é respeitado.

O formato de mensagem é descrito usando o formalismo Backus-Naur (Figura 3.9). O formalismo Backus-Naur é uma metalinguagem usada para descrição sintática de linguagens livres de contexto, protocolos de comunicação, entre outros (APPARAO et al., 2003; FIELDING et al., 1999; MCCRACKEN; REILLY, 2003). O formato de mensagem proposto permite a identificação do *subject*, o *tipo* deste *subject*, uma lista opcional de atributos que o caracterizam e uma lista opcional de *subject* internos, caso ele seja um *subject* composto, como acontece com um *CellularSpace* que possui vários *subjects* do tipo *Cell* em seu interior. Os *subjects* internos são recursivamente definidos no mesmo formato.

No *pipeline* de visualização desenvolvido, o protocolo é usado no módulo de aquisição, estágio em que a mensagem é construída (serializada) e no módulo de decodificação, etapa do processo em que a mensagem recebida é verificada e desserializada. Esses dois módulos implementam a sequência de transições estabelecidas pela máquina de estados finitos mostrada na Figura 3.10. O protocolo define uma forma padronizada e bem estabelecida de comunicação entre *subject* e *observers*. Além de permitir que estados completos ou parciais de *subjects* sejam visualizados. Até o momento, toda a informação necessária para a construção das visualizações é obtida via protocolo de comunicação e não é necessário o uso de metadados.

3.6 Interface de programação do *pipeline*

O *pipeline* de visualização possui duas funções em sua API - *Application Programming Interface* na linguagem TerraME: `Observer()` e `notify()`. Como é derivada de Lua, ela possui funções construtoras que são aquelas invocadas com a sintaxe `nomeMetodo{ atributo1 = valor1, atributo2 = valor2, ... }` para mimetizar a definição de classes em linguagens orientadas a objetos. No entanto, elas são interpretadas como chamadas de funções que recebem uma tabela de atributos como parâmetros: `nomeMetodo({ atributo1 = valor1, atributo2 = valor2, ... })`.

```
obsGrafico = Observer{
   type="chart",
   subject= celula1,
   attributes={"agua", "temperatura"}
}
```

Figura 3.11 – API de criação de observers na linguagem TerraME

O método construtor `Observer{...}` permite a instanciação de observadores de tipos pré-definidos no *framework* C++ de TerraME. O modelo da Figura 3.11 cria um *observer* do tipo gráfico que monitora os atributos **agua** e **temperatura** do *subject* **celula1**. O parâmetro `type` recebe o nome do tipo de observador que será instanciado, escolhido entre os diversos tipos disponíveis no *framework* C++ de TerraME. O parâmetro `subject` informa qual *subject* TerraME será monitorado e o parâmetro `attributes` recebe um lista de nomes dos que atributos serão visualizados.

```
1 -   cs = CellularSpace{ xdim = 4 }
2 -   cs:createNeighborhood{...}
3 -   cs:synchronize()
4 -
5 -   (...)
6 -
7 -   for t = 1, 100, 1 do
1 -       forEachCell(cs, rain )
8 -       forEachCell(cs, infiltration)
9 -       forEachCell(cs, superficial_flow)
10 -
11 -      (...)
12 -
13 -      cs:notify(t)
14 -  end
```

(a)

```
1 -   cs = CellularSpace{ xdim = 4 }
2 -   timer = Timer{
3 -   Event{ period = 0.5, action = function(event)
4 -       forEachCell(cs, rain )
5 -       forEachCell(cs, infiltration )
6 -       forEachCell(cs, superficial_runoff)
7 -
8 -   (...)
9 -
10 -  end},
11 -  Event{ period = 1, action = function(event)
12 -  cs:notify()      -- atualiza os observers do CellularSpace 'cs'
13 -  timer:notify() -- atualiza os observers do Timer 'timer'
14 -  end}
15 -  }
16 -  timer:execute(100)
```

(b)

Figura 3.12 – Diversos usos da função `notify()`.

Depois de criados, os observadores estão prontos para monitorar o *subject* a que foram associados. Nos locais do código do modelo em que o usuário do TerraME deseja visualizar o estado interno de um *subject,* basta ele inserir explicitamente uma chamada à função `notify()` deste *subject*, conforme a Figura 3.12. Desta maneira, todos os *observers* associados a ele irão automaticamente passar a exibir esse estado. A função `notify()` possui como parâmetro o tempo de simulação e é usado pelo TerraME *Observer* para

apresentar visualizações dependentes de tempo, como ocorre em gráficos de séries temporais onde se deseja visualizar a variação do valor de um atributo do modelo durante a evolução da simulação. Na Figura 3.12a o parâmetro `t` passado ao método *`notify()`* na linha 13 resulta na geração de uma série temporal de mapas que ilustram a dinâmica de um espaço celular declarado na linha 1. Na Figura 3.12b invoca a função *`notify()`* a partir de eventos de um objeto `timer`. O primeiro evento é executado pelo simulador a cada 0,5 unidade de tempo e simula a drenagem da água. O segundo evento apenas atualiza os *observers* de `cs` na linha 12 e de `timer` na linha 13. Desta maneira, as visualizações podem ser atualizadas em frequências menores que aquelas nas quais as regras do modelo são computadas.

Vários tipos de observadores foram implementados para o TerraME durante este trabalho. Contudo, podem não ser suficientes. Quando novos tipos de observadores são necessários, eles devem ser implementados por herança da classe *`Observer`* presente na *API C++* do *pipeline* (Figura 3.4). Assim, implementar o método abstrato *`draw()`* responsável por executar o mapeamento visual do *pipeline*. Novos tipos de *subject* também pode ser implementados por meio de herança da classe abstrata *Subject* e implementação de seu método abstrato *`getState()`* que implementa os módulos de aquisição do *pipeline*. A API C++ torna o *pipeline* extensível. Enquanto que a API Lua, permite que modelos implementados em versões anteriores do TerraME sejam facilmente instrumentados com código que permita a visualização da simulação.

3.7 Projeto das visualizações de modelos ambientais

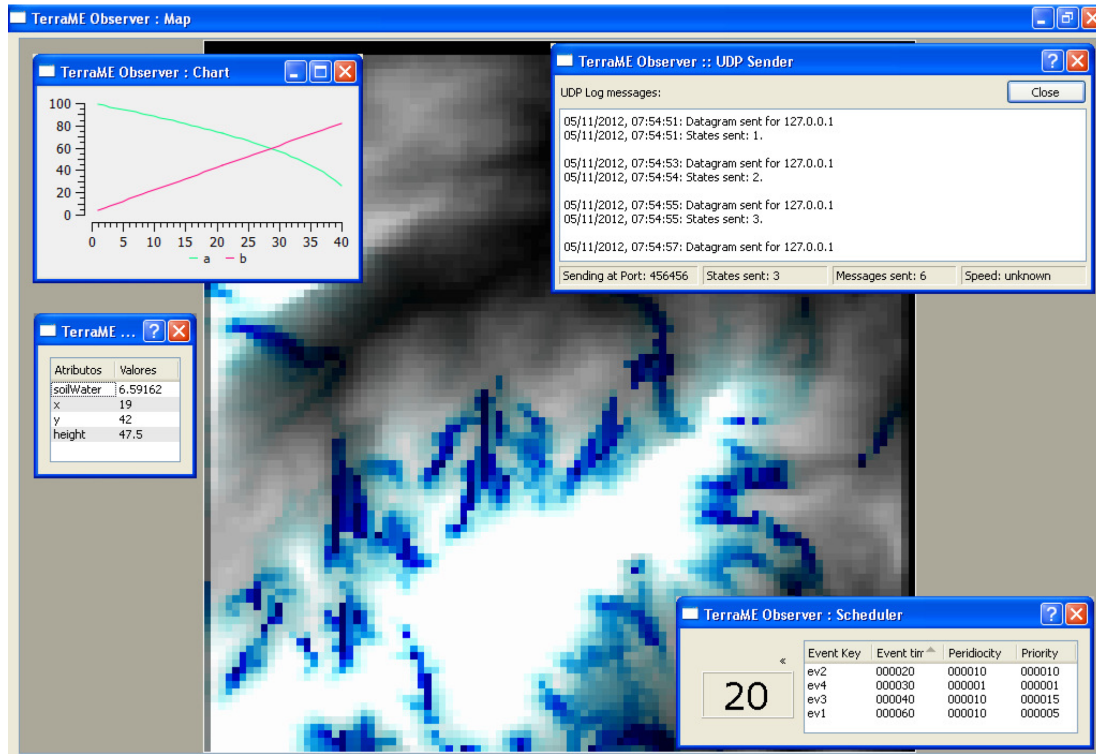


Figura 3.13 – Diversos tipos de observadores da plataforma de modelagem TerraME.

Para de facilitar o desenvolvimento de modelos ambientais em TerraME e, ainda, atender todos os requisitos funcionais identificados para um *pipeline* de visualização ambiental (Subseção 3.1), vários tipos de observadores foram desenvolvidos durante este trabalho. Alguns são ilustrados na Figura 3.13. Os observadores são capazes de apresentar as variáveis de estado contínuo e discreto, visualizar as dimensões de tempo, espaço e comportamento de modelos ambientais, exibir graficamente a coevolução de variáveis de estado.

Os *observers* básicos (*chart*, table, *logfile*, *textScreen* e transmissão via rede) estão disponíveis para todos os *subjects* TerraME. Enquanto que, outros *observers* só podem ser associados a determinados tipos de *subjects*, são eles: o observador *scheduler* definido apenas para o tipo *Timer*; os observadores *image* e *map* disponíveis apenas para tipos que representam relações espaciais; e observadores *state machine* disponíveis apenas para os tipos *Agent* e *Automaton*.

3.8 Seleção das métricas de desempenho

Entre as métricas de desempenho disponíveis para avaliar o *pipeline* de visualização desenvolvido neste trabalho, são utilizadas: o tempo de resposta do *pipeline*, o consumo de memória ocasionado pelas estruturas de dados que implementam o *pipeline* e o número de *bytes* transmitidos através dele. O tempo de resposta é definido como o tempo percebido pelo usuário desde o momento em que ele solicita a atualização da visualização e o instante em que a visualização termina de ser exibida. O consumo de memória avalia a sobrecarga imposta ao sistema de computação pelas estruturas de dados intermediárias mantidas pelo *pipeline.* O número de *bytes* transmitidos é útil para avaliar a carga de trabalho ao qual o *pipeline* foi submetido e para avaliar o consumo dos canais de comunicação disponíveis em *hardware*, isto é, consumo dos barramentos de dados ou e consumo da banda passante de rede.

O tempo de resposta do *pipeline* de visualização também pode ser visto como o tempo necessário para executar os módulos de aquisição, decodificação, mapeamento visual, *rendering* e exibição. É importante notar que o módulo de *rendering* é implemetado por tecnologias de computação gráfica que são utilizadas de forma *ad hoc* neste trabalho, como a biblioteca C++ Qt (http://qt-project.org) para *rendering* 2D e a biblioteca C++ OGRE (www.ogre3d.org) para *rendering 3D.* O desempenho do módulo *de rendering* não é influenciado por decisões de projeto tomadas neste trabalho, ele depende apenas da qualidade das placas gráficas instaladas no ambiente de teste e da implementação destas bibliotecas.

Logo, o tempo de resposta do *pipeline* de visualização pode ser decomposto em:

- Tempo de aquisição *(recovery time)*: Tempo gasto para acessar o valor dos atributos no espaço de memória da linguagem de alto nível, serializa-los seguindo um protocolo de comunicação (Subseção 3.5).
- Tempo de decodificação *(decoding time)*: Tempo necessário para desserializar e decodificar os dados preparando-os para o estágio de mapeamento visual.
- Tempo de mapeamento visual (*visual mapping time*): Tempo gasto para gerar a representação visual dos atributos monitorados, de maneira a atribuir cores, formas e coordenadas a elementos de dados.

- Tempo de *rendering (rendering time)*: Tempo consumido para que a biblioteca de renderização gere a imagem bidimensional a partir dos artefatos recebidos do módulo de mapeamento visual.
- Tempo de exibição (*display time*): Tempo necessário para mostrar as imagens renderizadas no dispositivo gráficos de saída (tela, plotter, impressora, etc);
- Tempo de espera (*waiting time)*: Na fase sequencial do *pipeline*, o tempo de espera inclui o tempo que as transformações aguardam pelo uso do processador sempre que ele está ocupado por outra transformação ou pelas atividades de controle do *pipeline*. Na fase paralela do *pipeline*, o tempo de espera incluiu o tempo que uma transformação permanece na *bag* de tarefas do mecanismo *bag-of-tasks* esperando para ser executada.

3.9 Definição do ambiente de teste

Experimentos podem ser influenciados por vários parâmetros como, as características do ambiente de teste, por exemplo, quantidade de memória RAM, qualidade da placa gráfica, desempenho e arquitetura do processador e, finalmente, velocidade de transmissão da rede em situações em que a simulação e visualizações ocorrem em equipamentos distintos.

Os experimentos realizados para avaliar o *pipeline* de visualização ocorreram em dois ambientes distintos. O primeiro foi composto por um computador monoprocessado AMD Athlon II X3, 2.7 GHz e 4GB de memória rodando Windows 7 64 bits. O segundo ambiente de teste, por um computador monoprocessado Intel Xeon E5620, 2.4 GHz e 32 GB de memória rodando Windows 7 64 bits. O processador AMD Athlon possui três núcleos (uma *thread* por núcleo) e cada núcleo possui 512KB de memória *cache* nível 2 (L2) dedicada. O processador Intel Xeon possui quatro núcleos (duas *threads* por núcleo) e cada núcleo possui 256KB de memória *cache* L2 dedicada. Ele também possui uma memória *cache* L3 de 12 MB compartilhada pelos quatro núcleos.

3.10 Planejamento dos experimentos

Tabela 3.1 – Carga de trabalho

Experimentos	**Atributos**	***Observers***	**No. de observadores**
Teste 1	3	*ObserverMap*	2
Teste 2	13	*ObserverMap*	12

Características da carga de trabalho a qual o *pipeline* é submetido também podem determinar seu desempenho: os tipos de *subject* e *observer*, o número de requisições de atualização por unidade de tempo, o volume de dados visualizado, etc. Dentre esses fatores, um subconjunto foi selecionado como parâmetros que iriam variar durante os experimentos: a quantidade de *observers*, o número de atributos do modelo que é observado.

Os experimento foram conduzidos para *subjects* do tipo *CellularSpace*, pois nos modelos ambientais os espaços celulares são, em geral, as representações que armazenam o maior volume de dados. Eles representam vastas regiões geográficas como uma grade de células, na qual cada célula (*Cell*) é caracterizada por diversos atributos. Além disto, na análise de modelos ambientais, uma das atividades mais comuns e interessantes é buscar compreender a dinâmica, distribuição e padrão espacial dos processos simulados. Por esta razão, o tipo de observerador *ObserverMap* foi utilizados em todos os experimentos. Esse tipo de visualização permite construir imagens bidimensionais a partir de dados recuperados do *CellularSpace*. Nos experimentos, o tamanho do *CelluarSpace* foi fixado em 10000 células.

Os experimentos consideraram um tempo de simulação de 100 iterações e foram repetidos 10 vezes cada para que estatísticas, como média e variação, das métricas de desempenho pudessem ser avaliadas. A cada iteração, a atualização dos *observers* era invocada e os tempos gastos em cada transformação, a quantidade de memória consumida e o número de *bytes* transmitidos eram anotados para posterior análise.

Foram conduzidos quatro diferentes conjuntos de experimentos para avaliar o impacto das decisões de projeto arquitetural do *pipeline* sobre seu desempenho. O primeiro conjunto de experimentos avaliou o uso do *blackboard* (BB) como memória intermediária. O segundo conjunto de experimentos avaliou o uso de dois protocolos diferentes para comunicação no *pipeline*: O protocolo desenvolvido neste trabalho e o protocolo chamado Protocol *Buffers*[3] *(ProtBuff)*. O *Protocol Buffers* é uma tecnologia de código livre desenvolvida pelo Google em substituição ao XML para a transmissão de estruturas de dados. Os dados foram estruturados através de uma *Interface Definition Language* – IDL que depois de compilada gera as classes que encapsulam esses dados. A IDL usada neste trabalho é mostrada no Apêndice A. O terceiro conjunto de experimentos avaliou o uso do paradigma *Bag of Tasks* (Bag) na

[3] http://code.google.com/protobuf/

implementação do paralelismo de *pipeline.* O quarto conjunto de experimentos avaliou a estratégia de transmissão apenas das mudanças sofridas pelos *subjects*.

Todos os conjuntos de experimentos foram dividido em Teste 1 e Teste 2, conforme Tabela 3.1. No Teste 1, cada objeto *Cell* do *CellularSpace* continha quatro atributos (**x**, **y**, **attr1** e **attr2**). Os atributos **attr1** e **attr2** juntamente com as coordenadas (**x**, **y**) das células a que pertenciam eram visualizados por *ObserverMaps* distintos. No Teste 2, cada célula continha treze atributos (**x**, **y**, **attr1**, **attr2**, **...**, **attr11**). Para cada tripla atributo e coordenada da célula, um *ObserversMap* distinto foi instanciado. Desta maneira, foi possível avaliar o impacto da visualização simultânea de múltiplos atributos de um mesmo *subject*.

Finalmente, um último conjunto de experimentos comparou o desempenho do serviço oferecido pela arquitetura de *pipeline* projetada e implementada neste trabalho, com o desempenho dos serviços oferecidos por outras plataforma de modelagem ambiental, especificamente, Repast, MASON e NetLogo. Para isto, os módulos de *rendering* da arquitetura foram implementadas em Qt[4]. Os códigos fontes dos ambientes de modelagem foram baixados de seus respectivos repositórios, investigados, os métodos que participam do serviço de visualização foram identificados e, posteriormente, instrumentados com instruções que registrassem as métricas de desempenho. Desta maneira, o mesmo modelo foi elaborado e desenvolvido em cada plataforma de modelagem ambiental (Apêndices B, C, D e E). Estes testes foram realizados somente no ambiente de teste AMD Athlon.

Neste último conjunto de experimentos, apenas a quantidade de mudanças sofrida pelo espaço celular foi considerada um parâmetro dos experimentos. Uma grade celular de 10000 objetos foi considerada em todos os experimentos. Devido a restrições da plataforma NetLogo quanto ao uso do objeto *world*, apenas uma visualização espacial foi usado em todos os experimentos.

Este último conjunto de experimentos também foi dividido em dois testes. No Teste 1, em que todo o espaço é alterado e no Teste 2, apenas metade do espaço sofre alteração. Os testes consideraram um tempo de simulação de 100 iterações e foi repetido 10 vezes. A cada iteração, a atualização da visualização era invocada e o tempo de resposta anotado. Desta

[4] http://qt-project.org

forma, a análise considerou apenas a média dos tempos de resposta dos serviços de visualização. Apenas para o ambiente TerraME, esse experimento foi expandido para avaliar o impacto de mudanças em porcentagens menores (25%, 12%, 6%, 3% e 1%) do *CellularSpace*.

4 RESULTADOS OBTIDOS

Esta seção apresenta os resultados obtidos pelo projeto em três direções: formação de recursos humanos, publicações e resultados científicos.

4.1 Formação de recursos humanos

Este trabalho contribuiu para a formação de dois alunos de Iniciação Científica do curso de Ciência da Computação da Universidade Federal de Ouro Preto e um título de mestrado obtido no curso de Pós-graduação dessa mesma instituição. O aluno Conrado Carneiro Bicalho com o projeto intitulado "Comunicação Eficiente entre Modelos Computacionais de Fenômenos Geográficos e Observadores em Realidade Virtual" e o aluno Patrick Brunoro Soares com o projeto "Realidade Virtual Aplicada a Análise de Modelos Computacionais de Fenômenos Geográficos", ambos tutoreados nos anos de 2012 e 2013. O título de mestre obtido em Agosto de 2013 com o nome "TerraME Observer: um *pipeline* extensível para visualização em tempo real de modelos espacialmente-explícitos".

4.2 Publicações

Duas publicações foram produzidas oriundas desse trabalho. Os resultados parciais foram publicados em (RODRIGUES; CARNEIRO; ANDRADE, 2012) e os resultados finais em (RODRIGUES; CARNEIRO; ANDRADE, 2013).

4.3 Resultados científicos

A partir dos experimentos realizados no decorre deste projeto para avaliar a arquitetura do *pipeline* de visualização desenvolvida, obtivemos os resultados apresentados nesta seção. As métricas de desempenho tempo de resposta, número de *bytes* transmitidos e consumo de memória foram consideradas.

Para melhorar estas métricas de desempenho, a cada ciclo de desenvolvimento do projeto, diversas técnicas, algoritmos e estruturas de dados foram gradativamente incorporadas à

implementação do *pipeline* e seu impacto avaliado. Os resultados apresentados a seguir ilustram os benefícios e desafios trazidos por cada uma destas soluções.

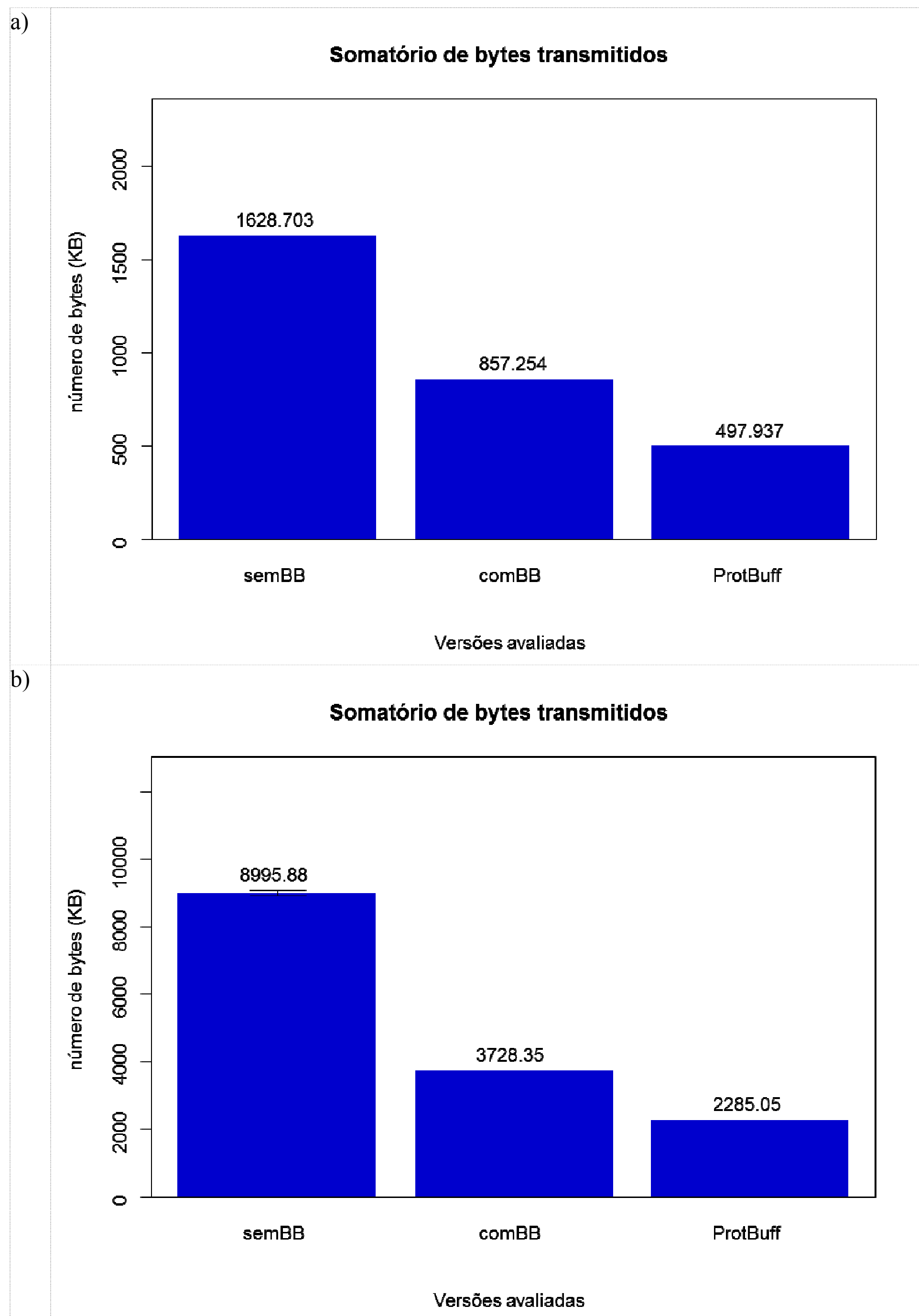


Figura 4.1 – Somatório de *bytes* transmitidos via *pipeline* avaliando o uso do *blackboard* (BB) e, em seguida, combinado o BB com o *Protocol Buffer* (ProtBuff) como estratégias para redução do número de *bytes* transmitidos. Em (a) os resultados do Teste 1 e em (b) os resultados do Teste 2.

A Figura 4.1 mostra que o uso do *blackboard* (BB) como memória *cache* para evitar a execução dos módulos de gestão de dados do *pipeline* reduziu significativamente a quantidade

de *bytes* transmitidos em cada notificação de mudança em, aproximadamente, 47% para o Teste 1 e 58% para o Teste 2. Essa técnica combinada com a tecnologia *Protocol Buffers* obteve um ganho, em torno de 69% para o Teste 1 e 74% para o Teste 2. A melhoria foi alcançada principalmente devido à praticidade do *Protocol Buffers*, pois não há necessidade de utilizar um *token* como delimitador entre os valores serializados como ocorre em nossa implementação e pela eficiente decodificação comprovada dessa tecnologia.

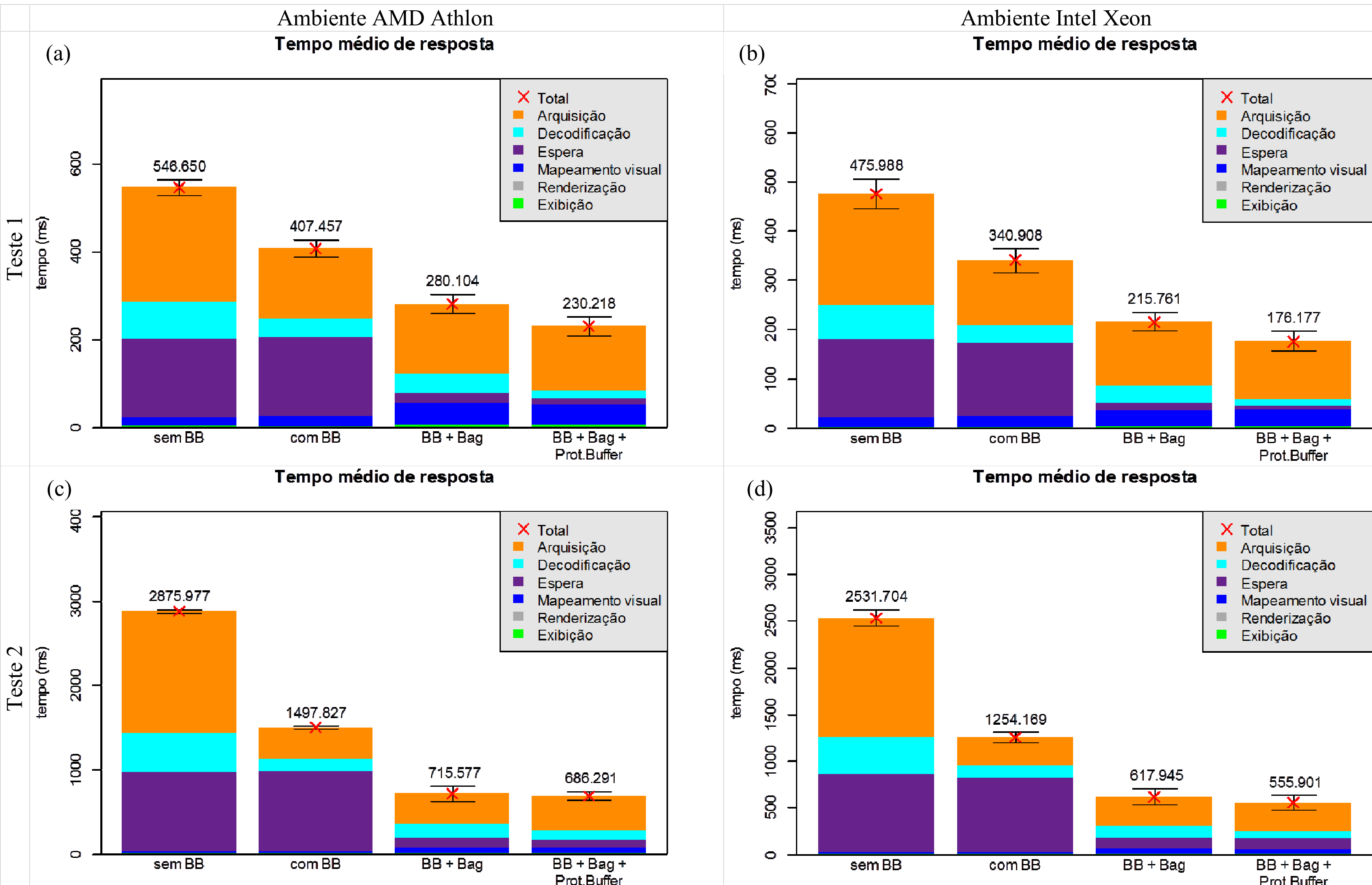


Figura 4.2 – Tempo médio de resposta da arquitetura TerraME *Observer*

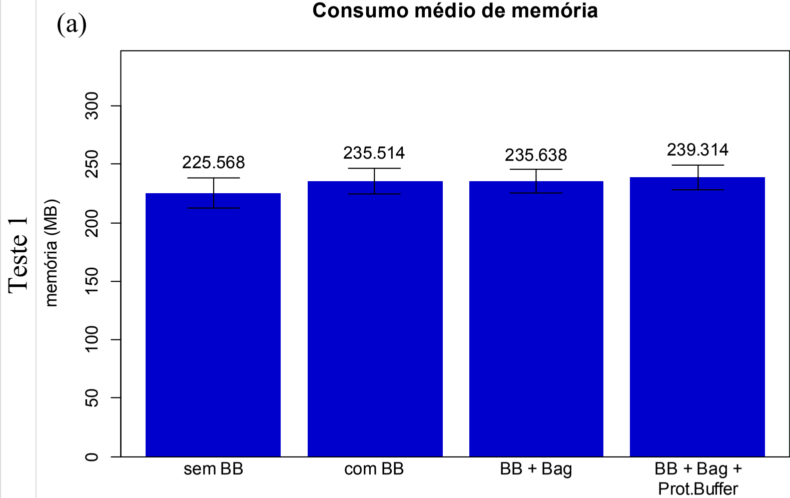
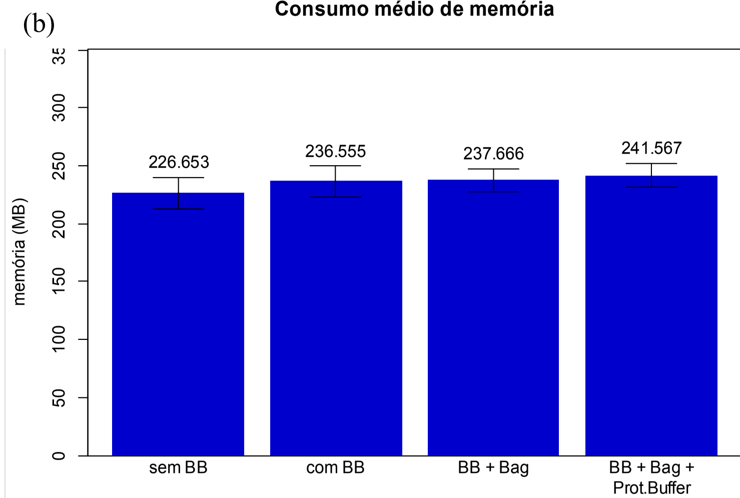
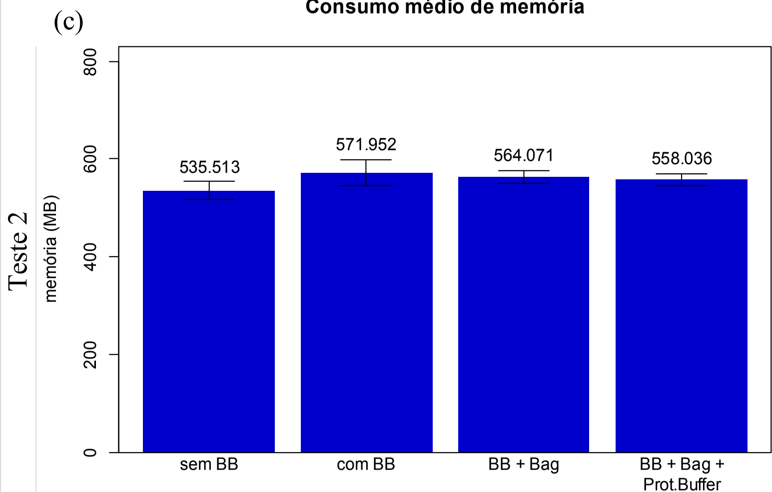
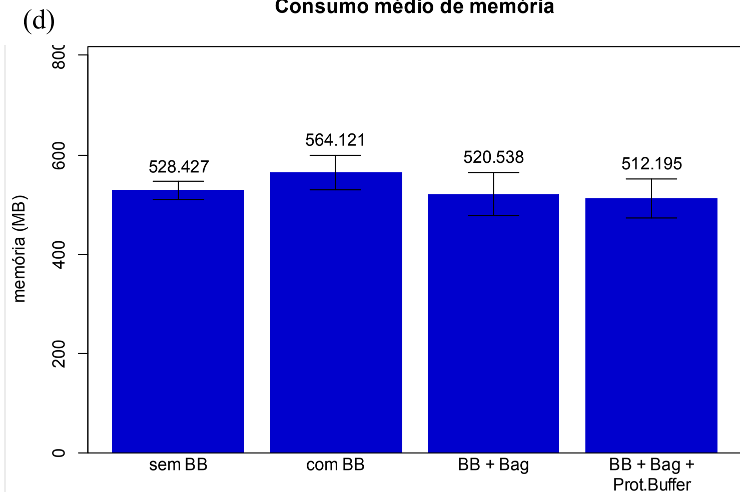
	Ambiente AMD Athlon	Ambiente Intel Xeon
Teste 1	(a) 	(b) 
Teste 2	(c) 	(d) 

Figura 4.3 – Consumo de memória

A Figura 4.2 apresenta o tempo médio de resposta do serviço de visualização provido pela arquitetura de *pipeline* desenvolvida neste trabalho. É notório que o tempo consumido pelo módulo de *rendering* é ínfimo quando comparado com o tempo consumido pelos demais módulos. O *blackboard* foi à primeira estratégia aplicada para reduzir o tempo gasto pela fase de aquisição e de decodificação dos dados no espaço de memória da linguagem Lua. A redução verificada nos ambientes de teste Athlon e Xeon foi de aproximadamente 40% na aquisição e 50% na decodificação para o Teste 1. Para o Teste 2 a redução foi de aproximadamente de 75% e 70% nos tempos de aquisição e de decodificação, respectivamente.

Após essas melhorias, a execução sequencial dos módulos do *pipeline* tornou-se o gargalo da arquitetura. Isto é evidenciado quando o tempo de espera das solicitações de atualização torna-se o maior componente do tempo de resposta do serviço de visualização (Figura 4.2 com BB). O mecanismo de *Bag of Tasks* foi utilizado para implementar o paralelismo de *pipeline* a partir do módulo de mapeamento visual e, desta forma, reduzir o tempo de espera. A redução no tempo de espera foi de aproximadamente 90% em todos os experimentos e em ambos ambientes de teste.

Posteriormente, o módulo de aquisição tornou-se novamente o gargalo do *pipeline* (Figura 4.2 BB + Bag). Então, a tecnologia *Protocol Buffers* foi avaliada como estratégia de redução do tempo de serviço desse módulo. No Teste 1 dos ambientes Athlon e Xeon, o *Protocol Buffers* reduziu em aproximadamente, 10% no tempo de aquisição e 55% no de decodificação. Porém, esta tecnologia foi projeta para transmitir com alto desempenho até 1 MByte de dados. Acima deste limiar, a tecnologia por apresentar um desempenho piorado. Por isto, no Teste 2 quando vários atributos são visualizados simultaneamente, o tempo de aquisição aumentou em aproximadamente 12% no ambiente Athlon e de 2% no ambiente Xeon. Contudo, o tempo de decodificação foi reduzido em todos os experimentos: aproximadamente 55% no Teste 1 e 30% no Teste 2 para o ambiente Athlon; e 60% no Teste 1 e 45% no Teste 2 para o ambiente Xeon. Embora o tempo de aquisição tenha aumentado no Teste 2, o tempo médio de resposta do serviço de visualização reduziu em aproximadamente 10% no ambiente Xeon e de 5% para o ambiente Athlon.

A Figura 4.3 apresenta o consumo médio de memória da arquitetura de *pipeline* em todos os conjuntos de experimentos realizados. O consumo aumentou até um limite máximo de

aproximadamente 6,5% devido ao armazenamento intermediário de dados realizado pelo *blackboard*. A pequena redução no consumo de memória apresentada com a adição do mecanismo de *Bag of Tasks* ocorreu devido à reorganização e reimplementação de grande parte das estruturas de dados e algoritmos de controle do *pipeline*.

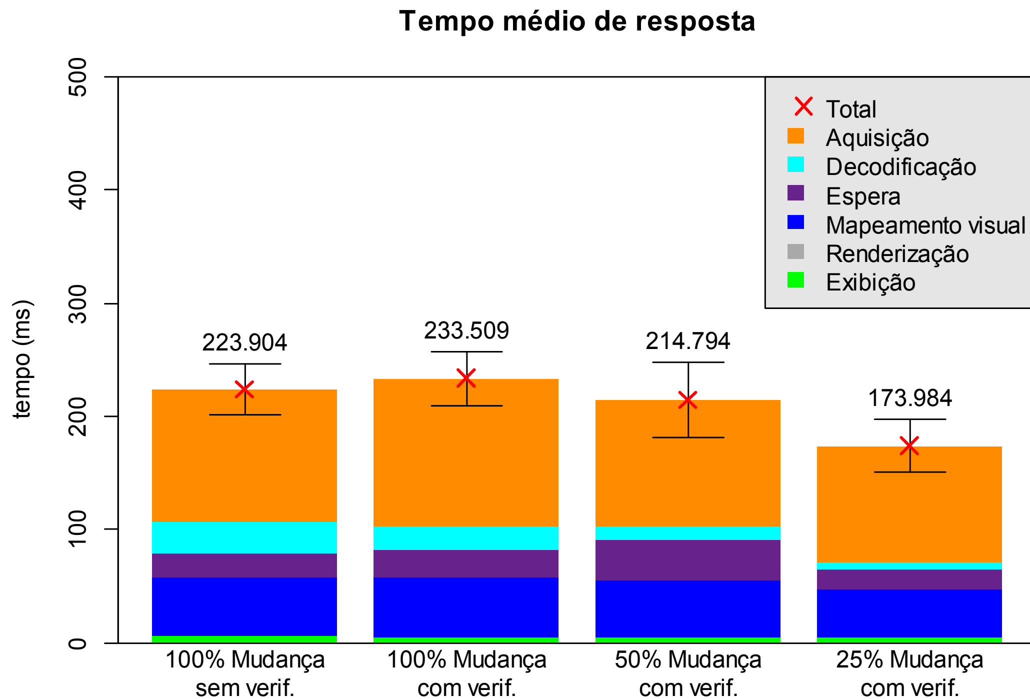


Figura 4.4 – Sobrecarga causada pelo algortimo utilizado para transmitir apenas mudanças do estado interno dos modelos.

A Figura 4.4 apresenta a sobrecarga gerada pelo uso do algoritmo utilizado para transmitir apenas mudanças, inferior a 5%. Para isto, o desempenho da arquitetura de *pipeline* proposta foi avaliada com e sem a incorporação deste algoritmo. Os experimentos para a análise de desempenho utilizaram o ambiente de teste Athlon e a carga de trabalho do teste Teste 1 (Tabela 3.1). Na Figura 4.4, os dois primeiros tempos à esquerda registram o tempo de resposta do *pipeline* sem e com o controle de mudanças, respectivamente. Os tempos de espera e de aquisição tiveram um incremento de 12% e 15%, respectivamente, devido a incorporação deste algoritmo. Contudo, o tempo de decodificação sofreu uma redução próxima de 30%, devido ao fato do número de *bytes* a ser decodificado ter diminuído em 40%, aproximadamente. Pois, mesmo que o estado interno (atributos definidos pelo usuário) de todas as células tenha mudado, alguns atributos (meta atributos) das células não mudam, como suas coordenadas e seus identificadores únicos. Por isto, quando o controle de mudança está ativado, estes últimos não são retransmitidos via *pipeline* a cada atualização.

Quando apenas 50% das mudanças foram transmitidas (Figura 4.4 50% Mudança com verif.), os tempos de aquisição e de decodificação reduziram em torno de 14% e 40%, respectivamente. O tempo de espera aumentou em 50%. Tornar o módulo de decodificação muito mais rápido, faz com que as notificações por atualização permaneçam mais tempo esperando por na fila entrada (*bag of taks*) do módulo de mapeamento. A fase de mapeamento visual teve uma redução aproximada de 7% devido ao menor volume de dados transmitido via *pipeline*.

Quando 25% das mudanças são transmitidas (Figura 4.4 25% Mudança com verif.), os módulos que consumiam mais tempo obtiveram reduções significativas em tempo de serviço: aquisição em 22%, decodificação em 67% e mapeamento visual em 20%. O tempo de espera reduziu em 30%.

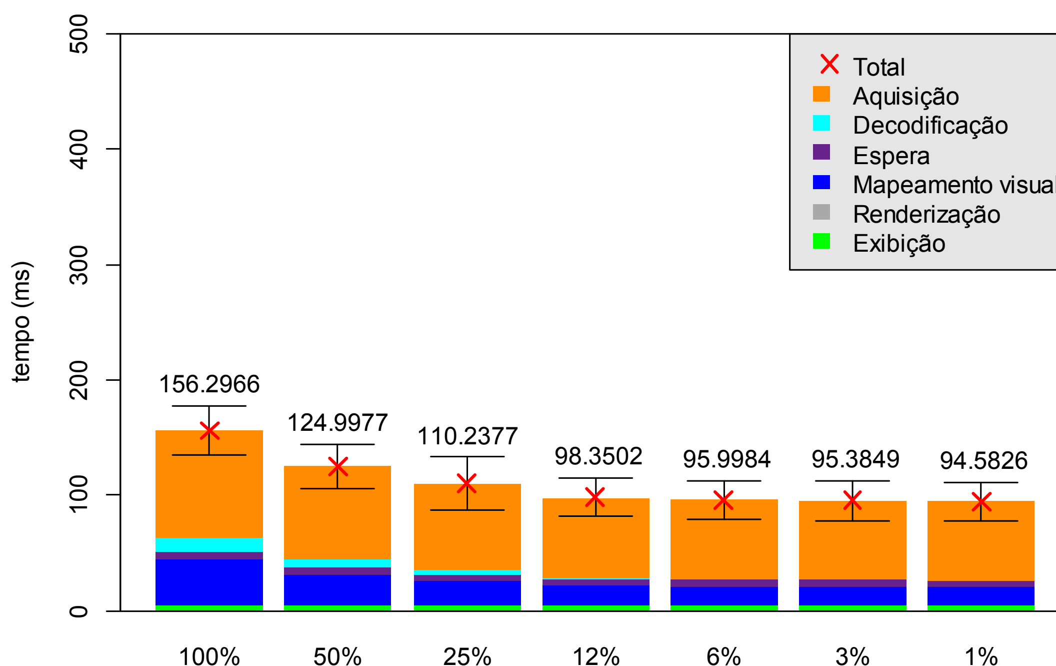


Figura 4.5 – Tempo de resposta vs. Porcentagem de variação

A Figura 4.5 apresenta uma comparação entre os tempos de resposta *versus* a porcentagem de alteração. Percebe-se que o desempenho da arquitetura desenvolvida é definido principalmente pelo tempo gasto na fase de aquisição. Quando a porcentagem de alteração nos atributos é inferior a 25%, o tempo consumido pela fase de espera, mapeamento visual e exibição tornam-se constantes e o tempo gasto na fase de decodificação é próximo à zero.

5 CONCLUSÃO

A visualização científica é útil para transformar dados em informações relevantes. Ela permite que hipóteses acerca destes dados sejam construídas e verificadas à medida que apoiam as atividades de análise e modelagem dos dados. Aplicada à modelagem ambiental ela contribui para a melhoria dos modelos ambientais, para o processo de tomada de decisão e para a definição de políticas públicas.

No contexto do Projeto URBIS Amazônia, este trabalho contribuiu para a visualização em tempo-real de sociedades de agentes e na melhor compreensão de suas relações e interações em um contexto geográfico. Esse projeto obteve resultados relevantes na formação de recursos humanos: tutoria de alunos de Iniciação Científica, um título de mestrado e publicações e na área acadêmica: desenvolvimento e avaliação de uma arquitetura de alto desempenho, extensível e flexível para a implementação de *pipelines* de visualização destinados ao monitoramento e análise em tempo-real de simulações ambientais.

REFERÊNCIAS

AGUIAR, A. P. D. et al. Modeling the spatial and temporal heterogeneity of deforestation-driven carbon emissions: the INPE-EM framework applied to the Brazilian Amazon. **Global Change Biology**, v. 18, n. 11, p. 3346–3366, 2012.

AGUIAR, A. P. D.; CÁMARA, G.; ESCADA, M. Spatial statistical analysis of land-use determinants in the Brazilian Amazonia: Exploring intra-regional heterogeneity. **Ecological Modelling**, v. 209, p. 169–188, 2007.

ALMEIDA, R. M. et al. **Simulando padrões de incêndios no Parque Nacional das Emas, estado de Goiás, Brasil**X Brazilian Symposium on GeoInformatics. **Anais**...Rio de Janerio, Brazil: 2008

ALMEIDA, R. M.; MACAU, E. E. N. **Stochastic cellular automata model for wildland fire spread dynamics**Journal of Physics: Conference Series. **Anais**...2011

AMARAL, S. et al. Comunidades ribeirinhas como forma socioespacial de expressão urbana na Amazônia: uma tipologia para a região do Baixo Tapajós (Pará-Brasil). **Revista Brasileira de Estudos de Populaćão (Impresso)**, 2013.

APPARAO, V. et al. **XML Protocol (XMLP) Requirements**World Wide Web Consortium, Note NOTE-xmlp-reqs-20030728, , 2003. Disponível em: <http://www.w3.org/TR/2003/NOTE-xmlp-reqs-20030728>

BEAUMONT, O. et al. Centralized versus distributed schedulers for bag-of-tasks applications. **IEEE Transactions on Parallel and Distributed Systems**, v. 19, n. 5, p. 698–709, 2008.

BENOIT, A. et al. Scheduling concurrent bag-of-tasks applications on heterogeneous platforms. **IEEE Transactions on Computers**, v. 59, p. 202–217, 2010.

BUSCHMANN, F. et al. **Pattern-oriented software architecture: a system of patterns**. New York, NY, USA: John Wiley & Sons, Inc., 1996.

CÂMARA, G. et al. Amazonian Deforestation Models. **Science**, v. 307, n. 5712, p. 1043–1044, 2005.

CÂMARA, G. et al. TerraLib: An open source GIS library for large-scale environmental and socio-economic applications. In: HALL, B.; LEAHY, M. G. (Eds.). **Open Source Approaches in Spatial Data Handling**. [s.l: s.n.].

CARDOSO, A. C. D.; VENTURA NETO, R. A evolućão urbana de belém: trajetória de ambiguidades e conflitos sócio-ambientais. **Cadernos Metrópole (PUCSP)**, v. 1, p. 55–76, 2013.

CARNEIRO, T. G. DE S. et al. An extensible toolbox for modeling nature-society interactions. **Environmental Modelling & Software**, v. 46, n. 0, p. 104–117, ago. 2013.

CHEN, C.; HRDLE, W.; UNWIN, A. **Handbook of Data Visualization**. Heidelberg, Berlin: Springer Berlin Heidelberg, 2008.

CORKILL, D. D. Blackboard systems. **AI expert**, v. 6, n. September, p. 40–47, 1991.

DEFANTI, T. A.; BROWN, M. D.; MCCORMICK, B. H. Visualization: Expanding Scientific and Engineering Research Opportunities. **IEEE Computer Society Press**, v. 22, n. 8, p. 12–25, 1989.

FIELDING, R. et al. **RFC 2616, Hypertext Transfer Protocol -- HTTP/1.1**RFC Editor, , 1999. Disponível em: <http://dl.acm.org/citation.cfm?id=RFC2616>

GAMMA, E. et al. **Design patterns: elements of reusable object-oriented software**. [s.l.] Addison-Wesley, 1995. v. 206p. 395

IERUSALIMSCHY, R. **Programming in Lua, Second Edition**. 2. ed. [s.l.] Lua.org, 2006.

IERUSALIMSCHY, R.; FIGUEIREDO, L. H.; FILHO, W. C. Lua - an extensible extension language. **Software - Practice & Experience**, v. 26, n. 6, p. 635–652, 1996.

JAIN, R. **The art of computer systems performance analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling**. 1. ed. Littleton, Massachusetts: Wiley, 1991. p. 685

KELLEHER, C.; WAGENER, T. Ten guidelines for effective data visualization in scientific publications. **Environmental Modelling & Software**, v. 26, n. 6, p. 822–827, jun. 2011.

LANA, R. M. et al. **Change Allocation in Spatially-Explicit Models for Aedes aegypti Population Dynamics**Simpósio Brasileiro de Geoinformática. **Anais**...2010

LANA, R. M. et al. Multiscale Analysis and Modelling of Aedes Aegyti Population Spatial Dynamics. **Journal of Information and Data Management**, v. 2, n. 2, p. 211–221, 2011.

LIMA, A. C.; SIMÕES, R.; MONTEMÓR, R. L. M. Espaço, cidades e escalas territoriais: novas implicações de políticas de desenvolvimento regional. **Economia e Sociedade (UNICAMP. Impresso)**, 2013.

LOPES, E. S. S.; NAMIKAWA, L. M.; REIS, J. B. C. **Risco de escorregamento: monitoramento e alerta de áreas urbanas nos municípios no entorno de Angra dos Reis - Rio de Janeiro**13o. Congresso Brasileiro de Geologia de Engenharia e Ambiental. **Anais**...São Paulo, Brasil: 2011

MARTORANO, L. G. et al. **Erosive Potential of Rains in the Climate Change Scenarios in the Upper Taquari River Basin, Brazil**Proceedings of the 2011 TROPENTAG: Conference on Tropical and Subtropical Agricultural and Natural Resource Management. **Anais**...2009Disponível em: <http://www.tropentag.de/2009/abstracts/links/Martorano_6nlK7jsx.php>

MCCORMICK, B. H. Visualization in scientific computing. **SIGBIO Newsl**, v. 10, n. November, p. 15–21, 1987.

MCCRACKEN, D. D.; REILLY, E. D. Backus-naur Form (BNF). In: **Encyclopedia of Computer Science**. Chichester, UK: John Wiley and Sons Ltd., 2003. p. 129–131.

MOREIRA, E. et al. Dynamical coupling of multiscale land change models. **Landscape Ecology**, v. 24, n. 9, p. 1183–1194, 2009.

MORELAND, K. A Survey of Visualization Pipelines. **IEEE transactions on visualization and computer graphics**, v. 19, n. 3, p. 367–378, 30 maio 2013.

MULLIGAN, M. **Environmental Modelling**. Chichester, UK: John Wiley & Sons, Ltd, 2004.

NII, H. Blackboard application systems, blackboard systems and a knowledge engineering perspective. **AI magazine**, v. 7, n. 3, 1986a.

NII, H. The blackboard model of problem solving and the evolution of blackboard architectures. **AI magazine**, v. 7, n. 2, 1986b.

NII, H. Y. **Blackboard Systems -- Report (Stanford University. Computer Science Dept.)**. [s.l.] Department of Computer Science, 1986c. p. 86

POPOVIC, M. **Communication Protocol Engineering**. Boca Raton, FL: CRC / Taylor & Francis, 2006. p. 453

REIS, J. B. C.; CORDEIRO, T. L.; LOPES, E. S. S. **Utilização do Sistema de Monitoramento e Alerta de Desastres Naturais aplicado a situações de escorregamento - caso de Angra dos Reis**14o. Simpósio Brasileiro de Geografia Física Aplicada. **Anais**...2011

RODRIGUES, A. J. C.; CARNEIRO, T. G. S.; ANDRADE, P. R. An Extensible Real-time Visualization Pipeline for Dynamic Spatial Modeling. **Journal of Information and Data Management**, v. 4, n. 2, p. 156–167, 2013.

RODRIGUES, A. J. DA C.; CARNEIRO, T. G. DE S.; ANDRADE, P. R. DE. **TerraME Observer: an extensible real-time visualization pipeline for dynamic spatial models**Simpósio Brasileiro de GeoInformática. **Anais**...Campos do Jordão, SP: 2012

SCHREINEMACHERS, P.; BERGER, T. An agent-based simulation model of human-environment interactions in agricultural systems. **Environmental Modelling & Software**, v. 26, n. 7, p. 845–859, 2011.

SPRUGEL, D. G. et al. Spatially explicit modeling of overstory manipulations in young forests: Effects on stand structure and light. **Ecological Modelling**, v. 220, n. 24, p. 3565–3575, 2009.

UPSON, C. et al. The application visualization system: a computational environment for scientific visualization. **Computer Graphics and Applications, IEEE**, v. 9, n. 4, p. 30–42, 1989.

VO, H. T. et al. Streaming-Enabled Parallel Dataflow Architecture for Multicore Systems. **Computer Graphics Forum**, v. 29, n. 3, p. 1073–1082, 12 ago. 2010.

VON NEUMANN, J. Theory of self-reproducing automata. **Mathematics of Computation**, v. 21, n. 100, p. 745, 1966.

WARE, C. **Information Visualization: Perception for Design**. San Francisco, CA, USA: Morgan Kaufman, 2004. p. 486

WOOD, J. et al. Using 3D in Visualization. In: DYKES, J.; MACEACHREN, A. M.; KRAAK, M.-J. (Eds.). **Exploring Geovisualization**. [s.l.] Elsevier, 2005. p. 295–312.

WOOLDRIDGE, M. J.; JENNINGS, N. R. Intelligent Agents: Theory and Practice. **Knowledge Engineering Review**, Knowl. Eng. Rev. (UK). v. 10, n. 2, p. 115–152, 1995.

WU, J.; DAVID, J. L. A spatially explicit hierarchical approach to modeling complex ecological systems: theory and applications. **Ecological Modelling**, v. 153, n. 1-2, p. 7–26, 2002.

ZEIGLER, B. P.; PRAEHOFER, H.; KIM, T. G. **Theory of Modeling and Simulation**. [s.l.] Academic Press, 2000. v. 132p. 510