Felipe Borges Alves
André Ferreira Bem Silva

Uma linguagem para programação concorrente e de tempo real com biblioteca voltada para desenvolvimento de jogos

Trabalho de Projetos II referente ao semestre 2009.2

Orientador:
Olinto José Varela Furtado

UNIVERSIDADE FEDERAL DE SANTA CATARINA

Bacharelado em Ciências da Computação

Departamento de Informática e Estatística

Florianópolis, 1 de dezembro de 2009

Felipe Borges Alves
André Ferreira Bem Silva

Uma linguagem para programação concorrente e de tempo real com biblioteca voltada para desenvolvimento de jogos

Bacharelado em Ciências da Computação

Departamento de Informática e Estatística

Florianópolis, 1 de dezembro de 2009

Trabalho de conclusão de curso defendido na Universidade Federal de Santa Catarina. Sob o título de “Uma linguagem para programação concorrente e de tempo real com biblioteca voltada para desenvolvimento de jogos”, defendido por André Ferreira Bem Silva e Felipe Borges Alves e aprovada em Florianópolis, 1 de dezembro de 2009 pela banca examinadora constituída pelos professores e convidados:

Olinto José Varela Furtado
Universidade Federal de Santa Catarina

Diego Dias Bispo Carvalho
Grupo Cyclops

Ricardo Silveira
Universidade Federal de Santa Catarina

José Mazzuco
Universidade Federal de Santa Catarina

“Dedico este trabalho a minha família, amigos e a Samanta Rodrigues Michelin por seu carinho, atenção e amor. Em especial, dedico-o para Pedro Bem Silva e Laura Ferreira Silva por sua compreensão e amor incondicionais"

– André Ferreira Bem Silva

Agradecimentos

- Dedicamos nossos sinceros agradecimentos para as seguintes pessoas:
  - Professor Olinto José Varela Furtado, pela dedicação e orientação desse trabalho
  - Diego Dias Bispo Carvalho, pelos auxílios, correções, amizade e compreensão
  - Tiago Nobrega, pelas correções, amizade e tempo dedicado
  - Jeferson Vieira Ramos, por correções no trabalho e amizade
  - Aos membros da banca pelas observações, correções e tempo disponibilizado
  - Aos professores do curso de bacharelado em Ciências da Computação, pela base teórica, sem a qual o trabalho não teria êxito
  - Aos caros colegas de curso, porque muitos participaram direta ou indiretamente do trabalho

“A humildade é a base e o fundamento de
todas as virtudes e sem ela não há nenhuma
que o seja."

– Miguel de Cervantes

Resumo

Esse trabalho descreve a proposta e implementação de uma linguagem que visa facilitar a programação concorrente e de tempo real. A linguagem proposta estende a linguagem padrão C, adicionando algumas construções inexistentes nela. A biblioteca padrão da linguagem possui funcionalidades de programação de jogos que inclui primitivas de renderização, carregamento de arquivos de objetos, passos de otimização de renderização, simulação física, detecção de colisão, áudio 3D e carregamento de shaders. Também é função dessa biblioteca prover as estruturas básicas de programação concorrente e o suporte a programação de tempo real por meio de funções que alteram a captura de exceções de tempo real. Essa extensão, denominada CEx, foi implementada no compilador Clang, o qual gera código para a Low Level Virtual Machine (LLVM). A partir dessa implementação desenvolveu-se a um ambiente para desenvolvimento de jogos, o qual foi testado com alguns protótipos simples.

Palavras-chave: linguagem concorrente, linguagem de tempo real, computação gráfica, desenvolvimento de jogos

Abstract

This work describes a language that aims to ease the programming of soft real-time and concurrent systems. The defined language extends the standard C language, adding some new constructions to it. The language standard library provides some graphical utilities for game development, object file loading, rendering optimization passes, builtin physical simulation, high-level collision detection, 3D audio and shader loading. It also provides basic concurrent programming structures and real-time support by some functions which changes real-time exception handling. This extension, a.k.a as CEx, was implemented as a extension of the Clang compiler that generates code for the Low Level Virtual Machine (LLVM). From this implementation was possible to provide an environment for game development, which was tested by some simple prototypes.

Keywords: concurrent language, real-time language, computer graphics, game programming

Lista de Algoritmos

Lista de Figuras

Lista de Tabelas

Sumário

1 Introdução

Esse trabalho encontra-se no contexto de linguagens voltadas para aplicações específicas. As linguagens possuem gramáticas formalmente descritas (em geral). Linguagens são utilizadas para facilitar a fase de desenvolvimento dando uma maior capacidade sintática ao programador para que não seja necessário trabalhar diretamente com código de máquina (Assembly). Nesse trabalho propomos o desenvolvimento de uma linguagem de programação concorrente e de tempo real genérica e de uma biblioteca voltada para o desenvolvimento de jogos, constituindo um ambiente de desenvolvimento próprio para esse fim.

Uma máquina virtual, é uma máquina que executa um Assembly próprio, denominado bytecode, fazendo com que o mesmo programa rode em inúmeras máquinas físicas sem que seja necessário recompilá-lo, porém, a máquina virtual deve poder ser compilada nessas máquinas. Devido a isso, o mais comum é que máquinas virtuais sejam implementadas em linguagens que gerem diretamente código de máquina. Um exemplo de especificação e implementação de máquina virtual pode ser observado no caso Smalltalk, visto em (Goldberg e Robson 1983).

Uma linguagem concorrente possui mecanismos de sincronização e paralelismo que facilitam o desenvolvimento de códigos que poderiam rodar fisicamente ao mesmo tempo, acessando os mesmos recursos, assim como capacidade de intercomunicação entre objetos concorrentes. Um exemplo é o Concurrent C, o qual é analisado em (Peter, Buhr e Sartipi 1996).

Uma linguagem de tempo real é uma linguagem que possui características que facilitam o desenvolvimento de softwares que possuem alta dependência do conceito tempo, isto é, seu correto funcionamento depende de eventos ocorrerem no exato momento em que deveriam. Os programas de tempo real podem ser definidos em duas grandes categorias: *hard real-time e soft real-time*. Os programas *hard real-time* são aqueles que se falharem apresentam grandes riscos para o sistema e/ou a vida de pessoas, enquanto os de *soft real-time* são aqueles que caso não funcionem corretamente não acarretam consequências catastróficas ao sistema e/ou a vida de pessoas. Esse tipo de software está relacionado essencialmente ao escalonamento e acionamento de tarefas em tempo real, como visto em Liu et al(Liu e Layland 1973).

Uma biblioteca básica de uma linguagem de programação deve conter as funções necessárias para que o programador utilizar para que ele não precise refazer as funcionalidades básicas para cada programa que vá escrever. Um exemplo de código que existe na biblioteca da maioria das linguagens é a função de imprimir caracteres no console que na linguagem C é a função printf. A biblioteca padrão de C encontra-se junto ao documento que define o padrão ISO C99(ISO 1999).

Uma biblioteca para desenvolvimento de jogos, conhecida como *graphics engine* ou motor gráfico, geralmente contém: primitivas gráficas e objetos de alto nível (retângulo, esfera...), carregamento de objetos de sistema (obj, md2/md3, collada...), animação, carregamento de shaders, otimizações de renderização (culling), tratamento de colisão otimizado e simulação física, assim como tudo aquilo que facilite o desenvolvimento genérico e específico (de acordo com a categoria) do jogo. Um exemplo de implementação dessas features é a engine O*bject-Oriented Graphics Rendering Engine* (OGRE)(Farias et al. 2005).

Uma *engine* deve ser interessante o suficiente para facilitar o trabalho do programador, oferecendo objetos primitivos de mais alto nível em relação àqueles básicos implementados pelas *Application Program Interface* (API) OpenGL ou DirectX. O intuito de codificar-se uma engine, no caso dos jogos, é facilitar a programação de jogos para o caso da empresa produzir vários, tornando o ambiente de desenvolvimento mais homogêneo. O processo detalhado de produção e comercialização observa-se em (Bethke 2003).

A linguagem visará prover facilidade de desenvolvimento ao programador, ao oferecer um sandbox com as funcionalidades de desenvolvimento de jogos e construções de paralelismo e tempo real presentes na sintaxe e biblioteca padrão da linguagem, de modo a não só facilitar a programação como também a facilitar a depuração de código.

Pretende-se também implementar a linguagem via o frontend Clang/LLVM(Clang 2009) que possui otimização de código gerado em seis níveis, assim como análise e otimização dinâmica de código provido pela máquina virtual LLVM, estratégia essa que possibilitaria modo interpretado de execução de modo simples, uma vez que isso já está disponível no ferramental da máquina virtual.

1.1 Objetivos gerais

Especificar uma linguagem que estenda C e que possua construções de paralelismo e tempo real. Implementar a linguagem de modo a torná-la retro-compatível com a sintaxe C e de modo a tornar possível intercomunicação concorrente entre objetos de alto nível.

1.2 Objetivos específicos

Pretende-se analisar as linguagens de tempo real e paralelismo existentes, de modo a conseguir modelar uma especificação formal para uma nova linguagem e uma possível implementação para ela, baseando-se em ferramentas e códigos já existentes.

1.2.1 Analisar as linguagens de tempo real e paralelismo existentes

Analisar as linguagens da literatura que possuam construções de suporte à programação de tempo real e paralelismo. Após feita a análise, será feito a especificação da linguagem de modo a levar as idéias analisadas em consideração, aplicando-as diretamente ao mundo de desenvolvimento de jogos.

1.2.2 Criar uma linguagem baseada na sintaxe de C

Modelar e implementar uma linguagem que estenda a sintaxe de C, com funcionalidades de programação concorrente e de tempo real, cuja biblioteca inclua primitivas e métodos próprios para computação gráfica e programação de jogos incluindo todas as ferramentas matemáticas, físicas e gráficas.

1.2.3 Prototipar a especificação formal

Após a especificação da gramática, prevê-se a implementação dessa via uma ferramenta adequada, ou seja, via um gerador de compiladores ou como uma extensão à algum frontend já existente. A ferramenta deve ser analisada previamente e possuir facilidades interessantes que facilite o trabalho de implementação da gramática formal e permitir a geração de código intermediário.

1.2.4 Fazer comparativos finais

Realizar uma análise comparativa em linhas e desempenho de código gerado pela linguagem e por códigos equivalentes escritos em C e/ou C++ puro, para observar a robustez, legibilidade e flexibilidade da linguagem especificada. Também poderá ser observado a eficiência da implementação da linguagem ao observar-se o quesito desempenho.

2 *Trabalhos Correlatos*

A linguagem a ser desenvolvida nesse trabalho é particularmente interessante pelo fato de unir áreas como computação gráfica, programação paralela e tempo real. Fez-se uma pesquisa de bibliografia de modo a não reinventar conhecimentos vigentes no meio científico, tornando o trabalho não só mais válido, como também mais avançado que aqueles no qual se baseia. O trabalho que inspirou esse é o da *engine* e linguagem de programação da empresa idSoftware, denominada Quake-C, cuja especificação detalhada pode ser encontrada em (Quake-C 2008).

Nesse capítulo abordam-se os principais trabalhos relacionados à esse, nas áreas de compiladores, máquinas virtuais, programação concorrente e tempo real. É necessário revisar os trabalhos relacionados a cada uma dessas áreas para especificar formalmente as construções da nova linguagem proposta.

2.1 Compiladores

Para gerar-se código para algum alvo é necessário fazer uma tradução do código fonte de alto nível para o código alvo. Quando esse processo de tradução é de uma linguagem de programação de alto nível[1], tem-se um compilador. O processo de compilação, compreender algumas etapas: análise léxica, sintática e semântica, geração de código intermedíario, otimização e geração de código de máquina. Essas etapas fazem parte das duas estruturas principais de um compilador: o *frontend* e o *backend*.

2.1.1 Frontend

As etapas associadas ao frontend são as análises léxica, sintática e semântica, a geração de código intermediário e a otimização desse. As três primeiras são obrigatórias, porém as relacionadas ao código intermediário não, uma vez que um compilador pode gerar diretamente código de máquina e esse código não precisa ser necessariamente otimizado. Um resultado importante

[1] Entenda-se aqui como todas as linguagens como C, Haskell, Java, C++...

da passagem do frontend é a *Abstract Syntax Tree* (AST) do programa que é necessária para fazer-se vários tipos de otimizações, principalmente as que são relacionadas à expressões. Na árvore, cada nodo é uma construção obtida a partir do código fonte. Ela é denominada abstrata porque nem todos os elementos da sintaxe estão presente explicitamente na árvore, como por exemplo o agrupamento de parênteses na árvore que é implícito. A partir da árvore é possível gerar-se então o código por meio de uma iteração por seus nodos.

2.1.2 Backend

O backend corresponde a etapa de geração do código alvo a partir das estruturas geradas no frontend como a AST. Nessa etapa é feita a alocação para os registradores reais da máquina, as otimizações dependentes de arquitetura e a ligação do código gerado de modo a gerar um executável. Quando a máquina alvo é uma máquina virtual, denomina-se *bytecode*, o código gerado pelo compilador que pode ser rodado na máquina.

2.2 Máquinas virtuais

As máquinas virtuais são amplamente utilizadas e o sucesso dessas deve-se ao fato de o programador na verdade estar executando seu código num sandbox provido pela máquina, o qual geralmente não oferece acesso direto aos recursos da máquina, tornando menor a probabilidade de ocorrerem problemas graves no programa e facilitando a depuração, pela possibilidade de erros mais inteligíveis.

As máquinas virtuais executam bytecode próprio gerado pelo compilador relacionado a linguagem, de modo a não ser necessário compilar programas escritos em Java, por exemplo, para cada arquitetura diferente que pretende-se utilizar o código, todavia deve ser possível compilar o código da máquina virtual na arquitetura alvo. Dentre as principais máquinas virtuais está a QVM(Phaethon 2003), que está diretamente relacionada a esse trabalho pelo fato de ser uma máquina virtual para a linguagem Quake-C, especialmente desenvolvida para desenvolvimento de jogos. Outra máquina interessante é o LLVM, que constitui-se não somente numa máquina virtual, mas sim um framework inteiro para otimização e análise de código.

2.2.1 Quake-C

O Quake-C possui máquina virtual própria, implementada pela equipe da idSoftware®. Utiliza-se o código intermediário do compilador LCC, otimizado para arquitetura do tipo RISC

Figura 2.1: Exemplo de jogo escrito na linguagem Quake-C(idSoftware 2005)

como entrada para a ferramenta *q3asm* que gera bytecode para a máquina virtual implementada por eles, denominada QVM. Essa máquina possui bytecode similar ao da LLVM, porém opera baseada em pilha.

Essa máquina virtual, que foi utilizada na criação do jogo Quake 3, da empresa idSoftware, possui uma infra-estrutura completa para programação de jogos o que a torna um excelente caso de estudo para o trabalho na parte referente ao desenvolvimento de jogos (instruções de carregamento de mapas, API gráfica, arquitetura de cliente/servidor, esquema de som...). Um exemplo de utilização da infra-estrutura da máquina pode ser observada na figura 2.1, a qual demonstra uma cena dentro do jogo Quake 3, escrito em Quake-C, utilizando a técnica de *raytracing*.

2.2.2 LLVM

Low Level Virtual Machine, ou simplesmente LLVM, é um framework para otimização e análise de código. O LLVM contém uma máquina virtual que interpreta um *bytecode* próprio, de baixo nível, do tipo SSA (*Static Single Assignment*) em relação a linguagens como C/C++. O que torna esse framework interessante são suas otimizações em tempo de compilação, ligação e execução, as quais são explanadas em (Lattner e Adve 2004).

As otimizações de tempo de compilação e ligação são aquelas já vistas em muitos fra-

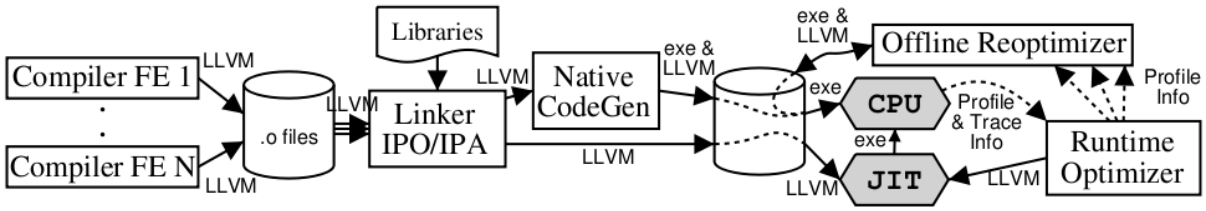

Figura 2.2: Arquitetura do LLVM(Lattner e Adve 2004)

meworks e compiladores, contudo elas possuem mais nível de escolhas quando comparadas as do GCC. As otimizações de compilação são oferecidas pelo frontend (e.g GCC), enquanto as de ligação fazem parte do framework LLVM diretamente, podendo por exemplo gerar código otimizado nativo para execução ou *bytecode* para a máquina virtual LLVM.

Para fazer otimizações em tempo de execução, ela precisa enxertar códigos leves no meio do código de execução. Aproveitando o tempo de cpu idle, ela pode fazer otimizações de performance específicas para um usuário (que roda o programa) e otimizar o código globalmente. Pode-se também gerar informações persistentes sobre o programa (informações obtidas de suas execuções).

O assembly da máquina LLVM é similar a um assembly CISC como a X86 e possui poucas instruções, devido a suas instruções poderem trabalhar com vários tipos de dados. Por exemplo, a instrução add pode fazer soma de qualquer tipo inteiro ou flutuante, reduzindo agressivamente o número de instruções necessários para a máquina. Essa simplicidade no bytecode interno, poderia ser aproveitada de modo a fazer uma máquina virtual que implementasse um assembly similar a esse, mesma abordagem dos desenvolvedores da QVM, abordada anteriormente.

Um fato interessante nas otimizações do LLVM é que a grande maioria delas é feita em cima do código intermediário interno, o qual é flexível o suficiente para implementar diversas linguagens como C/C++/Java, e simples o suficiente para ser interpretado rapidamente.

Outra facilidade do LLVM é o fato de que o código nativo pode ser gerado em tempo de ligação, de instalação ou de execução via um *Just in Time* (JIT) *Compiler* próprio, o que o torna uma das ferramentas mais flexíveis da área, com a vantagem de ser livre e bastante usada e patrocinada (e.g Apple®, Google®...). O maior desafio na utilização de tal é a curva de aprendizado necessária para utilização de sua API e linguagem intermediária. O processo de compilar, gerar e executar o código via LLVM é demonstrado na figura 2.2. Entre os frontends disponíveis para o LLVM está o Clang(Clang 2009) para linguagens da família C, o qual é descrito a seguir.

Clang

O Clang(Clang 2009) é um frontend único que compila C, C++, Objective-C e Objective C++, de modo a poder gerar código intermediário SSA para a máquina virtual LLVM. Atualmente apenas os frontends C e Objective-C encontram-se completamente prontos, enquanto os relacionados a C++ estão em desenvolvimento e instáveis ainda, devido a juventude do projeto e a complexidade da sintaxe dessa linguagem.

2.3 A linguagem C

A linguagem C foi implementada e desenvolvida no período entre 1969 e 1973. Seus criadores foram Dennis Ritchie e os Bell Labs. É uma linguagem criada sobre o paradigma procedural e feita para ser um "assembly portável", assim possui muitas facilidades de acesso à memória e torna simples trabalhar diretamente com o gerenciamento dessa, fazendo-a uma boa escolha para "programação de sistemas" (sistemas operacionais e embarcados), o que não é muito verdade em linguagens que escondem esse tipo de gerenciamento do programador como Java. O ISO mais atual da linguagem é o C99(ISO 1999), no qual esse trabalho baseia-se.

2.3.1 Principais características

- Características existentes
  - Sintaxe derivada da linguagem B, na qual o C baseou-se, assim como outras linguagens da época (ALGOL 68, FORTRAN...)
  - Tipagem fraca - variáveis de um tipo podem ser usados como de outro. Ex:. usar um caractere armazenado como short
  - Acesso e gerência de memória via ponteiros
  - Preprocessador para macros e inclusão de códigos (headers)
  - Biblioteca básica definida junto ao standard (libc)
- Características não existentes
  - Suporte a meta-programação
  - Tratamento de exceções
  - Nested functions
  - Suporte nativo para orientação à objetos

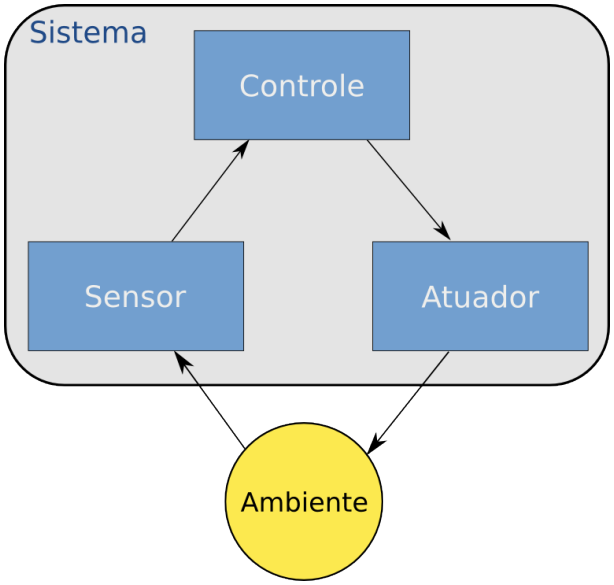

Figura 2.3: Arquitetura de controle tempo real para sistemas físicos (Buttazzo 2004)

2.4 Tempo real

Segundo Butazzo(Buttazzo 2004), softwares de tempo real são aqueles cuja funcionalidade depende diretamente do conceito de tempo, isto é, sua corretude não depende somente do resultado obtido, mas também do tempo em que esse resultado foram produzidos. Um sistema operacional de tempo real possui suporte a programação desse tipo de sistema.

Um dos conceitos mais importantes num software de tempo real é a previsibilidade, isto é, deve-se sempre saber tudo que se deveria saber sobre o software que está rodando, de forma que seja conhecido inclusive o tempo máximo no qual ele demora para fazer suas ações (pior caso). Outro conceito primário é o de *deadline*, que indica o tempo máximo que deve-se completar a execução de uma tarefa. A não completude dela dentro do tempo máximo esperado (*deadline*) acarreta que o software não está atrasado, mas sim errado (segundo a própria definição de tempo real) para o caso de aplicações críticas. De acordo com a influência sobre o ambiente de um programa não atingir uma *deadline*, os sistemas de tempo real podem ser divididos em dois tipos: *soft real-time systems* e *hard real-time systems*.

Uma arquitetura de controle de um sistema de controle físico genérico de tempo real pode ser observada na figura 2.3. Esse tipo de sistema é típico em várias aplicações como controle de injeção de motores, controle de estabilizadores de aeronaves, entre outros. Nessa arquitetura o sensor capta variações do ambiente que são repassadas ao controle, processadas e notificadas para o atuador que reage sobre o ambiente.

2.4.1 *Soft real-time systems*

Um sistema é dito *soft real-time* quando a perda da deadline não provoca danos sérios ao ambiente em controle e não compromete o correto funcionamento do sistema. Os casos onde a perda do deadline acarreta na perda da utilidade da computação são chamados de *firm*. Dentre as principais aplicações de *soft real-time systems* encontram-se:

- Gerenciamento de entradas (e.g teclado)
- Salvamento de dados persistentes (e.g relatórios/logs)
- Renderização em tempo adequado (e.g jogos/vídeos precisam manter um certo nível de quadros por segundo)

2.4.2 *Hard real-time systems*

Um sistema é dito *hard real-time* quando a perda da *deadline* provoca consequências catastróficas no ambiente que está sobre controle.

Quando um sistema operacional possui suporte à programação de *hard real-time* ele é dito um sistema operacional de *hard real-time*. Entre as atividades associadas aos sistemas desse tipo destacam-se:

- Controle baixo-nível de componentes críticos de sistema
- Aquisição de dados de sensores
  - Detecção de condições críticas
- Planejamento de ações sensor-motoras que interagem com o ambiente

2.5 Linguagens de tempo real

As linguagens de tempo real são aquelas que possuem suporte a programação de sistemas de tempo real, sejam eles soft ou hard realtime, diretamente na sintaxe ou biblioteca da linguagem. Entre elas estão o Real-time Concurrent C e a API e máquina virtual Real-time Java, explanados adiante.

2.5.1 Real-time Concurrent C

Dentro do Concurrent C, existe suporte a programação de programas de tempo real, como extensão a parte de concorrência. Ambas podem ser vistas em (Gehani 1991). A integração com tempo real é feita por meio de priorização das transações que acontecem entre os processos, que são as estruturas básicas da linguagem. Assim, existe um escalonamento de transações, feito explicitamente pelo programador, conforme demonstrado em (Gehani 1991). As transações da linguagem satisfazem a definição de Bloom para requisições de interação entre processos (*transaction* no Concurrent C), que define a aceitação da interação baseada nas seguintes informações:

- Nome da transação
- A ordem de recebimento de chamadas
- Argumentos da transação
- Estado do processo chamado

Caso dois processos comecem a trocar mensagem síncronas entre si, então eles entrarão em deadlock (ambos ficam aguardando mensagens). A solução para evitar esse tipo de deadlock é a troca de mensagens assíncronas, também presentes na linguagem.

2.5.2 Real-time Java

Real-time Java(Real Time Specification for Java) é uma extensão da plataforma Java com suporte a hard e soft real-time. Para que a linguagem Java seja *real-time*, é necessário que toda a máquina virtual apresente políticas de escalonamento e algoritmos característicos para esse tipo de programação. Dentre as características incluídas por Real-time Java destacam-se:

- Escalonador adequado a políticas real-time que possui o conceito de tarefas periódicas e esporádicas e suporte a deadlines
- Coletor de lixo próprio, demonstrado em (Bacon, Cheng e Rajan 2003) e modificado para atender os requisitos de tempo real. Assim, esse coletor de lixo diferentemente do coletor padrão do Java, possui tempo de coleta de lixo máximo conhecido (determinístico)

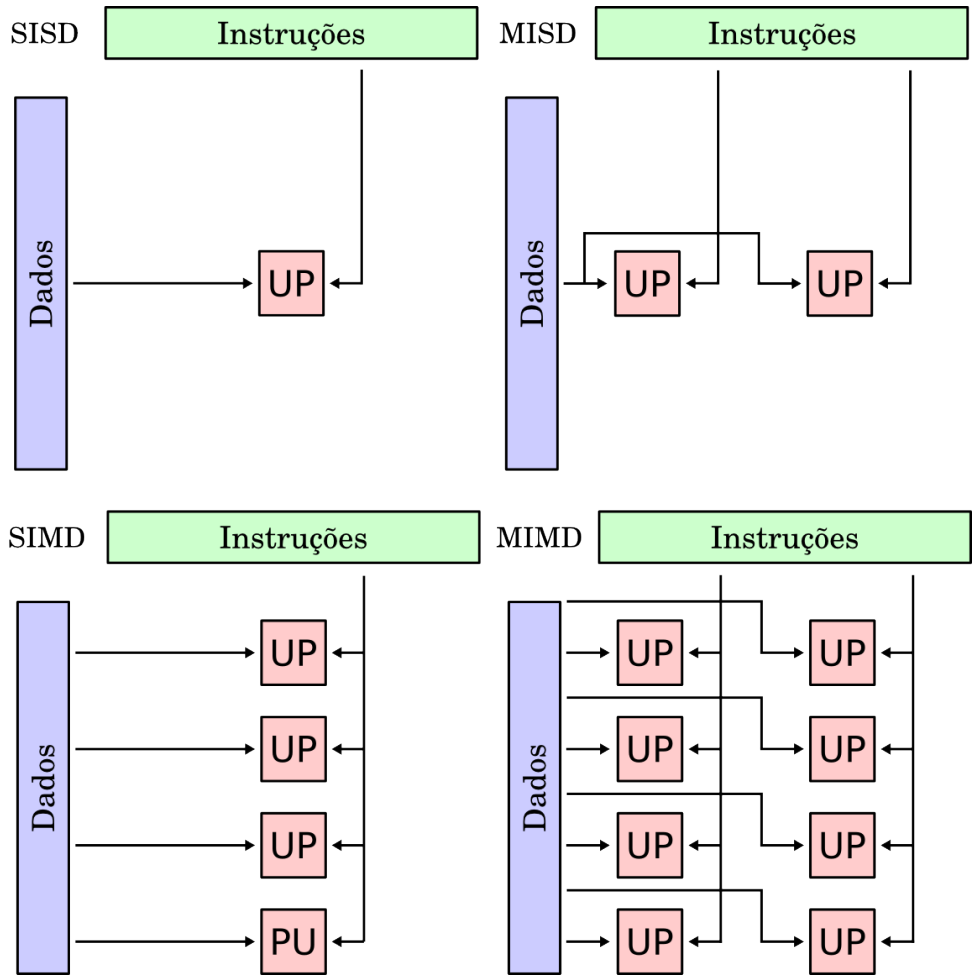

Figura 2.4: Taxonomia de Flynn e o acomplamento entre processadores e memórias

2.6 Taxonomia de Flynn

A taxonomia de Flynn(Flynn 1972) define o tipo de arquitetura de acordo com a quantidade de fluxos e controles concorrentes possíveis na arquitetura. A taxonomia define quatro grupos diferentes que podem ser observado na figura 2.4.

2.6.1 SISD

A família *Single Instruction Single Data* caracteriza-se por possuir somente controle e execução de um fluxo de código por vez. Não há concorrência real entre os programas, dado que cada programa irá executar de modo sequencial na única unidade de processamento (UP). Um exemplo real dessa arquitetura são todos os computadores Intel® da série Pentium.

2.6.2 MISD

Os *Multiple Instruction Single Data* definem uma arquitetura existente unicamente em teoria, pois esse conjunto é caracterizado por múltiplos fluxos atuando sobre os mesmos dados. Esse tipo de arquitetura teria algum tipo de aplicabilidade real, como quebra de chaves de segurança, porém na prática elas são um subconjunto de MIMD e não existem processadores comerciais implementados nessa arquitetura.

2.6.3 SIMD

Conhecidos como processadores vetoriais, os *Single Instruction Multiple Data* são capazes de atuar sobre grande quantidade de dados com uma instrução. Esse tipo de arquitetura foi amplamente usada na previsão climática e processamento de alto desempenho até a década de 1990. A partir do desenvolvimento de técnicas e evolução dos sistemas distribuídos, eles foram sendo substituídos por esses (que são MIMD).

2.6.4 MIMD

Multiple Instruction Multiple Data é a denominação dada a todos os multicomputadores e multiprocessadores. Os multicomputadores são caracterizados por possuírem uma rede de interconexão, normalmente especializada, que permite desempenhar uma computação distribuída entre vários computadores. Já os multiprocessadores, são aqueles que possuem múltiplos núcleos (UPs) interconectados por um barramento físico direto, de forma estática. A principal diferença entre ambos é que os multicomputadores possuem maior escalabilidade, uma vez que o número de processadores possíveis são da casa de milhares, enquanto nos multicomputadores são da casa de dezenas. Os MIMDs rodam em paralelismo real e o tipo de programação relacionada a eles é conhecida como programação paralela (ou concorrente) que é estudada na próxima seção.

2.7 Programação concorrente

A área de programação concorrente abrange todas as técnicas relacionadas a programação de entidades que concorrem paralelamente para terminar um trabalho, sendo a análise de regiões críticas do código o maior desafio nessa área, como pode ser visto em (Tanenbaum, Sharp e A 1992). Existem linguagens específicas para programação de códigos concorrentes, como Concurrent

C e Concurrent C++. Algumas dessas linguagens e suas idéias serão discutidas aqui de modo a trazer para a linguagem a ser implementada as características mínimas necessárias a todas as linguagens que forneçam suporte a esse tipo de programação.

2.7.1 Concurrent C

Essa linguagem foi definida por Narain Gehani e William D. Roome, conforme visto em (Peter, Buhr e Sartipi 1996), nos laboratórios AT&T da Bell em 1984. É uma extensão de C, ou seja, compatível com ela e foi feita para ser flexível o suficiente para rodar em um só computador, em uma rede ou mesmo em sistemas distribuídos com memória compartilhada. A idéia principal da linguagem é que existem vários processos leves (como threads) que são executadas em paralelo, trocando mensagens via *transações,* as quais podem ser síncronas ou assíncronas.

Ela não possui gerência de memória compartilhada e possibilita obter informações de processos, como estado, ID de cada processo concorrente. Oferece também o conceito de transação com tempo, a qual tem um temporizador que se "estourado", libera o processo chamador, ou seja, faz com que não hajam deadlocks devido a erros de mensagem não entregue, comuns em protocolos de comunicação concorrentes nos quais isso não fosse tratado.

2.7.2 Concurrent C++

Apesar do nome, essa linguagem não segue o ISO C++ rigidamente. É na verdade uma extensão da linguagem Concurrent C, usando construções de dados de C++ (classes) para atingir um patamar maior de abstração. Como visto em (Gehani e Roome 1988), foram necessárias algumas mudanças na linguagem base para adaptar a idéia de abstração de dados junto a idéia de concorrência. Algumas das idéias para melhorar essa integração na linguagem incluem:

- Variáveis tipo classe devem poder ser usadas no corpo de processo. Deve-se chamar o construtor do objeto ao criar o processo e o destrutor ao terminar o processo.
- Tipos de classe devem poder ser utilizados como argumentos para transações, argumento para processos e como retorno de transações.
- Providenciar capacidade de garbage collection
- Integrar a linguagem CC e CC++ em uma só

Algoritmo 2.1: Exemplo de utilização do for paralelo Cilk++

```
1 cilk_for (int i = 0; i < n; ++i) {
      c[i] = a[i] + b[i];
  }
```

2.7.3 Cilk++

A linguagem Cilk++(Cilk++ 2009) é uma extensão de C++ para programação concorrente. A idéia dessa linguagem é paralelizar facilmente um programa serial. A extensão da linguagem C++ consiste na adição de três palavras reservadas, o mesmo programa escrito sem o uso destas é um programa C++ válido com a mesma semântica. As palavras reservadas são:

- cilk_for - Executa um for paralelizando as iterações.
- cilk_spawn - Executa uma chamada de função de forma assíncrona, sem que seja necessário esperar a função retornar para continuar a execução.
- cilk_sync - Funciona como uma barreira para todos as funções executadas com cilk_spawn.

2.7.4 OpenMP

OpenMP(Dagum e Menon 1998) é uma API de programação paralela para C, C++ e Fortran. O paralelismo é indicado pelo programador usando diretivas de pré-processamento. O algoritmo 2.2 contem um exemplo de utilização da API OpenMP.

Como é possível ver no exemplo, OpenMP oferece uma grande facilidade na paralelização de programas, entretanto, fica evidente que com a utilização de uma linguagem específica é possível tornar a codificação mais simples. O algoritmo 2.1 demonstra o mesmo exemplo escrito utilizando cilk_for da linguagem Cilk++.

Algoritmo 2.2: Exemplo de utilização do for paralelo OpenMP

```
1 #pragma omp parallel for
  for (int i = 0; i < n; ++i) {
      c[i] = a[i] + b[i];
  }
```

Algoritmo 2.3: Exemplo de intercomunicação entre processos com Erlang

```
-module(message_sending).
-export([start/0, ping/2, pong/0]).

ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping finished~n", []);
ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_PID).
pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start() ->
    Pong_PID = spawn(message_sending, pong, []),
    spawn(message_sending, ping, [3, Pong_PID]).
```

2.8 Erlang

A linguagem Erlang foi especificada por Barklund e sua utilização em aplicações distribuídas confiáveis em presença de erros foi discutida por Amstrong et al (Armstrong 2003). Entre suas principais características estão:

- Todas construções são tratadas como processo
- Processos são isolados
- Transmissão de mensagens são a única maneira de comunicação
- Processos tem nomes únicos
- Se você conhece o nome do processo, então você pode mandar à ele uma mensagem
- Processos não compartilham recursos
- Tratamento de erros não é local
- Processos terminam a tarefa com êxito, caso contrário falham

Um exemplo de comunicação entre processos nessa linguagem pode ser observado no algoritmo 2.3, onde os dois processos exemplificados (ping e pong) trocam mensagens um certo número de vezes.

2.9 Computação gráfica

A parte de computação gráfica (CG) que compreende o trabalho trata-se de todas as bibliotecas de renderização, simulação física, carregamento de imagens, detecção de colisão, carregamento de shaders, incluindo também técnicas de otimização de renderização.

Ao fazer-se um estudo das construções utilizadas em computação gráfica, decidiu-se que a linguagem não teria construções específicas para ela, porém teria uma biblioteca que suprisse toda necessidade para desenvolvimento de aplicações de CG e desenvolvimento de jogos, pelo fato de não haver construções que fossem tão utilizadas de modo a incorporá-la a linguagem final.

As principais necessidades de suporte nas bibliotecas de aplicações de CG são matrizes quadradas, abstrações de view, suporte a transformações de rotação, translação e escalonamento, resolução de sistemas lineares, funções de suporte de cálculos vetoriais, abstrações de objetos básicos como linhas, triângulos, quadrados, polígonos, mesh de triângulos, entre outros. Também são necessárias funções de acesso e modificação de texturas, assim como sua aplicação em objetos complexos com vários tipos de mapeamento, suporte a esquema de iluminação (e.g *Phong Reflection*) e o carregamento de shaders para sobrepor funcionalidade padrão. Essas funções são todas explanadas no livro *Real-Time Rendering*(Akenine-Möller, Haines e Hoffman 2008).

Uma parte interessante de ter um código gráfico interpretado está exatamente na capacidade de poder-se realizar transformações arbitrárias, uma vez que pode-se inserir código dinamicamente. Pode-se daí, observar o resultado e acompanhar seu efeito, como visto em (Chen e Cheng 2005). A implementação e depuração de um editor de mapas seria facilitada pelo fato de ser possível executar comandos gráficos não presentes no código em tempo de execução.

2.9.1 APIs gráficas

As APIs gráficas mais usadas são o OpenGL® e o Direct3D®, os quais são interfaces que permitem uma maior flexibilidade ao programador, uma vez que esse não precisa-se trabalhar com comandos gráficos da placa de vídeo alvo diretamente, mas sim com primitivas de mais

alto nível que serão traduzidas de alguma forma para primitivas específicas da placa de vídeo (pelo driver utilizado).

A necessidade de APIs gráficas vem da década de 80, quando cada fabricante possuía suas próprias primitivas gráficas 2D e 3D, o que tornava custoso abstrair a funcionalidade do programa criado da implementação gráfica da desenvolvedora. Nesse contexto heterogêneo surgiu o OpenGL, idealizado pela Silicon Graphics Inc.® (SGI), e feito *opensource* para conquistar grandes empresas da época que utilizavam-se de um outro padrão, chamado de *Programmer's Hierarchical Graphics System* (PHIGS).

Hoje em dia, o OpenGL é um padrão mantido por consórcio internacional (Khronos) de muitas empresas e amplamente difundido em um grande número de dispositivos e sistemas operacionais.

O Direct3D, API proprietária da Microsoft®, foi desenvolvido junto ao Microsoft Windows® e possui versão para esse desde a versão Windows 95® do sistema operacional.

OpenGL

O *Open Graphics Library* é uma API *opensource* para renderização de primitivas gráficas, acelerada por hardware via drivers (em geral proprietários) de algumas desenvolvedoras que dão suporte à API. O standard inicial previa mais de 140 funções, citadas em (Board et al. 2007). Atualmente o padrão pode ter até 250 funções diferentes, encontradas em (Segal e Akeley 2008).

O OpenGL é multi-plataforma, isto é, é uma especificação independente de plataforma e possui versões para WIN32 e POSIX. Isto torna-o especialmente interessante para esse trabalho, uma vez que espera-se de uma linguagem que ela e sua biblioteca básica sejam independente de plataforma. Considerando-se que a biblioteca da linguagem dependa de uma API gráfica para ser passível de implementação, OpenGL é a escolha mais natural.

Direct3D

Direct3D é uma API, desenvolvida e mantida pela Microsoft, para renderização de primitivas gráficas, e faz parte do pacote DirectX®, desenvolvido pela mesma.

O Direct3D encontra-se na atualidade confinado ao ambiente WIN32, e portanto não faria sentido utilizá-lo pelas razões já descritas, contudo, talvez faça sentido permitir que seja utilizado ele quando o ambiente de execução seja WIN32, isto é, uma vez que o Direct3D é amplamente utilizado para desenvolvimento de jogos, pode ser mais simples tornar possível a

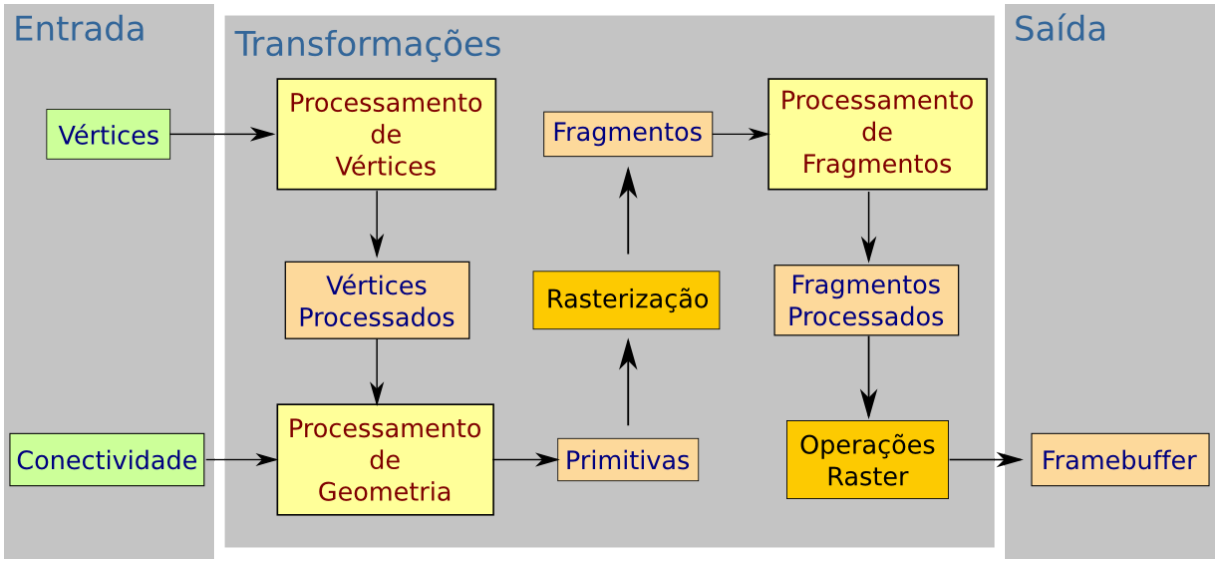

Figura 2.5: Pipeline da GPU

utilização dele, caso o programador assim queira.

2.9.2 Shaders

Shaders são programas que rodam no processador da placa de vídeo, i.e , uma GPU. Existem três tipos principais de shaders relacionados ao pipeline da placa de vídeo (figura 2.5) : *Vertex Shader* (VS), *Geometry Shader* (GS) e o *Fragment Shader* (FS), respectivamente, sendo essa a sequência em que eles são executados.

O VS é um shader que trabalha com os vértices e torna possível modificar suas coordenadas de textura e vértice, cor entre outros atributos.

O GS permite alterar as estruturas de dados (malhas) e gerar geometrias proceduralmente, assim como alterar coordenadas de vértice, textura e cor de vértice, assim como permite também iterar e atuar sobre os dados da malha.

O FS permite calcular o valor individual de cada pixel e é a etapa que ocorre após a rasterização, tornando essa etapa interessante, pois nesse momento os vértices e valores estão todos interpolados. A seguir é feita uma descrição da linguagem Cg (NVIDIA®), da linguagem GLSL (padrão do OpenGL), e da HLSL (padrão do Direct3D).

NVIDIA Cg

A linguagem Cg(Mark et al. 2003) é uma linguagem de alto nível, proposta pela empresa NVIDIA, a qual transforma um código com sintaxe similar a C em um assembly para OpenGL

(assembly ARB) ou DirectX, de acordo com a vontade do programador, desse modo abstraindo a necessidade de programar-se diretamente em uma linguagem de shading particular como era feito antes desse tipo de solução existir. O compilador Cg, também é flexível e aceita o código de ambas linguagens (HLSL e GLSL).

GLSL: OpenGL Shading Language

GLSL é uma linguagem de shading similar a Cg, principalmente quanto a sintaxe que tenta imitar C, com modificações é claro. Até a versão 3.0 do OpenGL, somente o FS e VS eram suportados pela API sem extensão. A partir do OpenGL 3.0(Segal e Akeley 2008) o GS será suportado também. A parte interessante de GLSL refere-se a sua generalidade e sua atuação multiplataforma como o padrão OpenGL. A linguagem e suas construções específicas são descritas em (Rost 2006). Muitas das instruções built-in da linguagem, como aquelas para cálculos vetoriais são interessantes a serem observadas e incluídas na implementação como biblioteca da linguagem alvo desse trabalho. O *Geometry Shader* tem suporte em GLSL por meio de uma extensão, para versões anteriores ao OpenGL 3.0 que trás para o GLSL as operações de bitwise que não existiam até então. O novo padrão pode ser observado em (Kessenich, Baldwin e Rost 2008).

HLSL: High-level Shader Language

HLSL é uma linguagem proprietária desenvolvida pela Microsoft para utilizar-se com Direct3D. Sua sintaxe é similar a NVIDIA Cg e ela foi desenvolvida paralelamente à essa. Como no GLSL possui 3 tipos de Shader diferentes, *o Vertex Shader, o Geometry Shader e o Pixel Shader* (equivalente ao *Fragment Shader* no OpenGL). HLSL é muito utilizado no meio de desenvolvimento de jogos, assim como o Direct3D. Alguns exemplos de utilização de HLSL para desenvolvimentos de jogos podem ser observados em (Luna 2003).

3 *Definições Preliminares*

Esse capítulo contém a especificação formal da nova linguagem proposta. Essas especificações levam em consideração os trabalhos revisados e as necessidades relacionadas (programação de tempo real, concorrente e gráfica) a esse trabalho, que propõe uma extensão ao C, contudo poderia ser também feita sobre alguma outra linguagem de sintaxe similar. A essa extensão, denominamos CEx (C Extended).

A seguir segue a proposição de várias estruturas da linguagem, uma biblioteca nova, voltada para o desenvolvimento de jogos, estruturas de dados, que são definidas pelo fato de não existirem em algumas bibliotecas básicas como a do C, e novas estruturas sintáticas.

3.1 Novas estruturas de dados básicas

Uma primeira análise sobre o C e vê-se que ele não possui biblioteca padrão de estruturas de dados, o que não é vantajoso, principalmente para programação de jogos onde essas estruturas são muito usadas. Assim a linguagem provê ao programador uma API de estruturas de dados básica, baseada na sintaxe de C++ e Java para manipulação de dados. Os novos tipos introduzidos na linguagem são: TreeSet, HashSet, List, Vector, HashMap e TreeMap.

As estruturas que não são de mapeamento recebem o tipo dos dados armazenados em tempo de compilação, conforme demonstrado no algoritmo 3.1.

Algoritmo 3.1: Exemplo de *bubble sort* com Vector

```
Vector<int> toSort = DS_vector_create();
...
DS_vector_addInSequence(10, 20, 1, -8, 1024, -4, -16, 32);
...
for (unsigned int i = 0; i < DS_vector_getSize(toSort); ++i)
    for (unsigned int j = 0; j < DS_vector_getSize(toSort) - 1; ++j)
        if (DS_vector_get(toSort, j) > DS_vector_get(toSort, j + 1))
            DS_vector_swap(toSort, j, j + 1);
```

<data_struct_type_specifier> ::= <type_specifier> | <pointer type_specifier> | CONST <pointer type_specifier>

<data_struct_type> ::= '<' <data_struct_type_specifier> '>'

<data_struct> ::= LIST <data_struct_type> | VECTOR <data_struct_type> |

HASH_MAP <data_struct_type> | TREE_MAP <data_struct_type> |

HASH_SET <data_struct_type> | TREE_SET <data_struct_type>

3.2 Construções de concorrência

Como falicidade para programação de códigos concorrentes são necessárias instruções de controle de acesso e sincronia de variáveis e instruções de paralelização de blocos de código. Essas funcionalidades foram divididas em cinco diferentes instruções, explanadas a seguir. São elas: *synchronized, async, parallel, parallel_for e atomic*. Para as seções seguintes as seguintes produções em notação BNF são interessantes de serem conhecidas:

<function_definition> ::= <function_definition_sync> | <function_definition_async>

<function_definition_sync> ::= <declaration_specifiers> <declarator declaration_list> <compound_statement>

<function_definition_sync> ::= <declaration_specifiers> <declarator compound_statement>

<function_definition_sync> ::= <declarator declaration_list> <compound_statement>

<function_definition_sync> ::= <declarator> <compound_statement>

Obs:. As produções utilizadas e não especificadas encontram-se disponíveis para observação nos anexos.

3.2.1 synchronized

Sincronização automática do statement alvo para acesso a ele com múltiplas threads de modo a tornar exclusividade o acesso ao bloco. Desse modo, em qualquer momento da execução, somente uma Thread poderá ter acesso a região crítica.

No exemplo de produtor-consumidor, demonstrado no algoritmo 3.2, observa-se a utilização da instrução para dois elementos. Nesse exemplo primeiro será realizado um *wait* na

Algoritmo 3.2: Exemplo de produtor-consumidor escrito em CEx

```
#include <inst/parallel.h> /*instructions include*/
#include <sync/sync.h> /*mutex and condition includes*/
#include <stdio.h> /*printf include*/

synchronizer arrayMutex, produceCondition;
const unsigned int ARRAY_MAX_SIZE = 1024;
unsigned int currentSize = 0;
int numberArray[ARRAY_MAX_SIZE], sum = 0;

void main() {
    arrayMutex = TM_createMutex();
    produceCondition = TM_createMutexCondition(arrayMutex);

    async (LONG_JOB) {//consumer
        while (1) {
            synchronized (produceCondition, arrayMutex) {
                while (currentSize) sum += numberArray[currentSize--];
            }
        }
    }
    while (1) //producer
        synchronized (arrayMutex)
            if (currentSize < ARRAY_MAX_SIZE)
                numberArray[currentSize++] = rand();
            else parallel { printf("Current sum %d\n", sum); }
}
```

variável *condition* e depois obtém-se a trava do *mutex*, respeitando-se a ordem em que o programador passou as variáveis.

Funciona também como mecanismo de sincronização geral, podendo ser utilizado com barreiras, semáforos, *conditions* e *mutexes*.

<synchronized_statement> ::= SYNCHRONIZED ’(’ <identifier_list> ’)’ <statement>

<statement>::= <synchronized_statement>

3.2.2 async

Modificador de funções e blocos de código, para executar em alguma thread específica. Indica que o bloco pode ser executado assincronamente, isto é, sem nenhuma relação com o bloco atual e portanto não há nenhum controle quanto ao conhecimento de término ou quanto ao estado da execução do bloco. Também não há nenhum tipo de troca de informação por meio da instrução.

<function_definition_async>::= ASYNC ’(’ IDENTIFIER ’)’ <function_definition_sync>

<parallel_statement> ::= ASYNC ’(’ IDENTIFIER ’)’ <statement>

<statement>::= <parallel_statement>

3.2.3 parallel

A palavra chave parallel define que o bloco indicado será executado em paralelo, ou que obtenha informações relacionadas à execução do bloco paralelizado, ao contrário do async que indica um código cuja semântica estabelce independência de dados e nenhum tipo de troca de informação por meio da instrução.

No exemplo do produtor-consumidor a keyword paralel é usada para realizar uma impressão no console do sistema assíncronamente. Observa-se que não há nenhuma garantia quanto a quando o código dentro do corpo do parallel irá executar, porém sabe-se que ele será escalonado para tal eventualmente caso não haja deadlocks no programa.

<function_definition_async>::= PARALLEL <function_definition_sync>

<parallel_statement> ::= PARALLEL <statement>

<statement>::= <parallel_statement>

3.2.4 parallel_for

Estrutura de repetição “for” onde as iterações são executadas em paralelo. Não há nenhum tipo de tratamento de dependência entre as iterações executadas em paralelo nesse caso. Caso seja necessário tratar as dependências o programador deve fazer o código manualmente utilizando-se das capacidades das bibliotecas envolvendo a estrutura reservada Task. Essa instrução baseia-se em muito no for paralelizado do OpenMP(Dagum e Menon 1998), porém com uma sintaxe mais simples.

<iteration_statement> ::=

PARALLEL_FOR ’(’ TYPE_NAME IDENTIFIER ’:’ IDENTIFIER ’)’ <statement> |

PARALLEL_FOR ’(’ TYPE_NAME IDENTIFIER ’:’ ’[’ CONSTANT ’,’ CONSTANT ’]’ ’)’ <statement> |

PARALLEL_FOR ’(’ expression_statement expression_statement ’)’ statement |

PARALLEL_FOR ’(’ expression_statement expression_statement expression ’)’ statement

Algoritmo 3.3: Exemplo de utilização de diversas construções da linguagem

```
#include <stdio.h>

static int TRUE = 1;
static Mutex m = MUTEX_INITIALIZER;

atomic int a = 2;

int main() {
    int i;

    /**Get idle thread from mixer that
     * only runs when there is an idle CPU*/
    int threadId = TM_getIdleJob();

    async (threadId) {
        for (i = 0; i < ITERATIONS_N; ++i) ++a;
    }

    parallel parallel_for (i = 0; i < ITERATIONS_N; ++i) a += 2;

    while (TRUE) printf(current value %d\n, a);

    return 0;
}
```

3.2.5 atomic

Garante o acesso atômico a variáveis[1] de modo a tornar operações aritméticas, binárias, lógicas e de comparação atômicas, sempre que possível. O comportamento de atomic para operadores que não são unários e não fazem parte do conjunto multiplicação, divisão, soma e diferença é indefinido.

<type_qualifier> ::= ATOMIC

3.3 Construções de tempo real

As instruções de tempo real que serão demonstradas mais adiante tem como objetivo fornecer suporte sintático na linguagem para programação de softwares de soft real-time. Assim, é necessário integrar dois tipos diferentes de blocos na linguagem: o bloco que só será executado uma vez e possui uma deadline específica e o bloco que executa periodicamente dentro de uma deadline. O suporte para o bloco de execução única é adicionado via *do_rt*, enquanto o suporte a blocos periódicos é feito via a instrução *periodic_rt*.

[1] Uma dentre {unsigned char, char, usigned shor, short, unsigned int, int, unsigned long long, long long}

3.3.1 do_rt

Essa instrução indica que o bloco alvo deve ser executado dentro de um tempo pré-estabelecido (*deadline*). Desse modo um bloco que seja indicado com essa palavra reservada deve executar com tempo máximo constante, conforme especificado. O comportamento padrão, caso a exceção de tempo real não seja tratada pela Task ou Thread executante, é de abortar o programa, funcionalidade similar a um sinal de *kill* do padrão POSIX, cuja especificação básica pode ser vista em (IEEE 2004). Para tratar a exceção de tempo real deve-se especificar qual o tratador da Thread/Task corrente. Caso a Task e a Thread tenham ambas tratadores instalados, então somente o da Thread será chamado.

<rt_statement> ::= DO_RT '(' <constant_expression> ')' <statement>

<statement> ::= <rt_statement>

3.3.2 periodic_rt

A semântica e sintaxe são idênticas ao do do_rt, a diferença está no fato que o bloco é tratado como periódico e deve ser executado antes do fim da *deadline* apontada, para cada iteração. Similar à utilizar-se um while com um do_rt como statement (similar porém não igual, uma vez que o while poderia executar mais de uma vez dentro da deadline, por exemplo). A primeira expressão constante refere-se ao tempo de deadline e a segunda à periodicidade do bloco.

<periodic_rt_statement> ::=

PERIODIC_RT '(' <constant_expression> ')' '(' <constant_expression> ')' <statement>

<statement> ::= <periodic_rt_statement>

3.4 Biblioteca básica

Para obter-se uma implementação mais simples de uma linguagem, muitos elementos que poderiam estar na sintaxe estão na verdade em sua biblioteca básica. Isso é feito para que a gramática da linguagem não torne-se demasiadamente grande e difícil de ler, estender e consertar.

Linguagens como C e C++ possuem bibliotecas básicas consolidadas para funções cotidianas, as quais não faria sentido implementar cada vez que fossem ser utilizadas, pelo fato de serem excessivamente comuns e difundidas em vários programas. Entre elas estão as funções

matemáticas, processamento de strings, entrada e saída (ex: console, arquivos...) entre outras funções básicas implementadas por basicamente toda linguagem de alto nível.

Como a linguagem deverá possuir suporte a primitivas básicas de renderização, a biblioteca básica dela possuirá também funções desse tipo, adicionalmente suporte não sintático para expressão de paralelismo e dependência de dados, isto é, estruturas básicas como *semáforos, mutexes, barreiras, monitores,* estarão disponíveis para utilização também.

As outras necessidades comuns à programação de jogos, que são carregamento de objetos, shaders, mapas, e todo tipo de entrada/saída desse gênero também possui suporte na biblioteca. Essas funções são típicas desse tipo de programação, mas elas também podem ser utilizadas para outros fins. Um exemplo possível seria utilizar-se das instruções de shader para fazer *GPGPU*, cujas principais características podem ser observadas em (Luebke et al. 2004).

A API de cada elemento das bibliotecas relacionadas podem ser observadas na seção de anexos, organizadas por áreas afins.

4 *Prototipagem*

Para implementar-se a especificação formal da linguagem é necessário ou criar uma ferramenta inteira, desde o começo (método custoso), ou utilizar alguma ferramenta previamente implementada e estendê-la. Nesse capítulo são apresentadas as alternativas nesse sentido e é definido um rumo para a implementação da nova linguagem, demonstrando o porquê das escolhas feitas. Demonstra-se também testes utilizando a implementação da linguagem, a arquitetura interna dos elementos utilizados e por fim um *First Person Shooter* (FPS) simples utilizando as construções e bibliotecas base da linguagem.

4.1 Discussões

Ao levar-se em consideração as ferramentas disponíveis e as possibilidades de implementação da linguagem, muitas perguntas surgem e para tomar-se uma decisão final, é razoável explicar porque não foi decidido outro meio. Nesta seção, serão discutidas as decisões feitas para implementação da linguagem. A seguir seguem as discussões de porque usamos C, a razão por não usarmos um gerador de compiladores e a motivação para escrevermos um motor gráfico próprio.

4.1.1 Por que C?

C é uma linguagem que, embora amplamente utilizada para programação de sistemas, possui muitas fraquezas, uma vez que é feita para ser simples e muito portável. Assim, existem muitos programas codificados em C e que continuam sendo mantidos. Para esse trabalho, talvez fizesse mais sentido utilizar-se da sintaxe de C++, uma vez que possui suporte nativo para orientação à objetos e meta-programação, assim como outras facilidades inexistentes em C.

Devido a complexidade da linguagem, suas implementações são extensas, e o único compilador de código aberto que forneceria infra-estrutura para implementarmos a extensão na atualidade seria o GCC. Por motivos já observados, esse compilador não seria de interesse desse

trabalho e o Clang, na atualidade, encontra-se incompleto quanto à sintaxe e semântica de C++, não sendo nem possível averiguar a corretude de um código alvo, quanto mais gerar algum tipo de código intermediário (ou de máquina).

4.1.2 Por que não um gerador de compiladores?

Yacc, ou Yet another compiler-compiler, é um programa para especificação de compiladores, no qual é possível especificar a parte sintática e semântica (feita artesanalmente) do código por meio de construções de sua linguagem, similar a notação simples de gramática. Na notação Yacc é possível misturar-se código C/C++ com a especificação formal (para gerar código automaticamente).

O Yacc não possui gerador de analisador léxico próprio e esse serviço é fornecido pelo Lex(Lesk e Schmidt 1975) que mistura também sintaxe C/C++ com especificação formal, nesse caso, expressões regulares. Tanto o Lex quanto o Yacc tem soluções atuais de mesma sintaxe básica (Bison/Flex).

Existem muitas outras ferramentas que podem ser citadas e seriam relevantes para o grupo de serem cogitadas a serem utilizadas como o GALS (Gerador de Analisadores Léxicos e Sintáticos)(Gesser 2002) e AntLR(Parr e Quong 1995), os quais poderiam ser utilizados, contudo preferiu-se estender o Clang (implementado através do método decendente recursivo), ao invés de usar um gerador de compilador dentre os citados, para gerar a linguagem CEx.

4.1.3 Por que um motor gráfico próprio?

A motivação para utilizar-se de motor próprio está no fato da demonstração de conhecimentos obtidos e a Ronin estar validada para a parte gráfica. Sabe-se que não seria o caso se fosse utilizado um motor como OGRE(Farias et al. 2005), pelo menos com respeito à validação de conhecimentos obtidos durante a graduação. Em (Matheson 2008) demonstram-se razões para codificar-se uma *engine* própria.

4.2 Parsing e geração de código

O protótipo de implementação da nova linguagem necessitará de algum tipo de compilador (frontend) para gerar código executável em alguma máquina (virtual ou não), o qual é definido nessa seção.

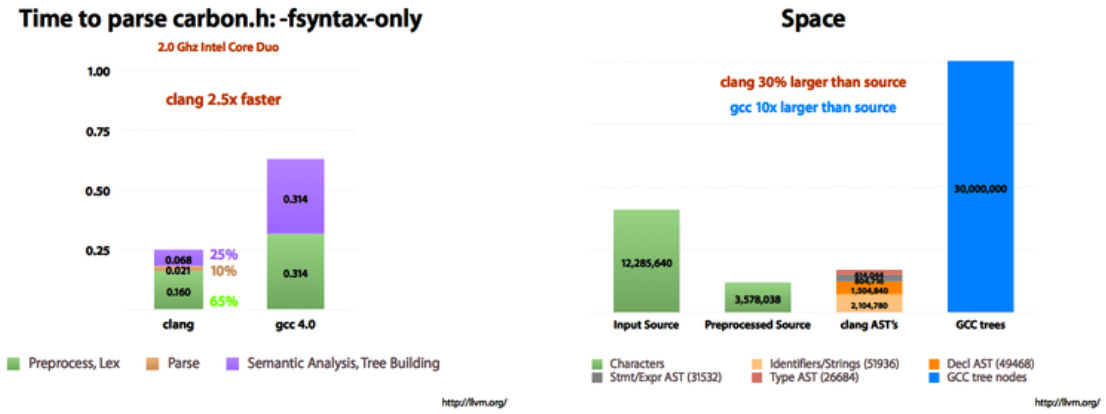

Figura 4.1: GCC vs Clang. Exemplo de parsing do Carbon.h(Clang 2009)

Para implementar a nova linguagem, decidiu-se estender um compilador C, ao invés de usar um gerador de compiladores. Assim, o compilador escolhido foi o Clang, que faz parte do projeto LLVM, conforme visto no capítulo 2. A seguir discute-se algumas características desse projeto que nos fizeram escolher prototipar a linguagem por ele.

O parsing e a geração serão feitos via Clang, o qual possui drivers de compatibilidade com o gcc e tenta ser mais expressivo nos diagnósticos de erros (principalmente em C++), assim como na utilização de memória em relação ao tamanho do código, cujas estatísticas do site deles demonstram ser 30% do tamanho do código de entrada, enquanto no GCC tende a ser 1000% maior, pelo menos para o header Carbon.h, que faz parte da API do MacOS/X, o qual inclui cerca de 12 MB de código fonte. Tais dados podem ser observados na figura 4.1.

O parsing no Clang é implementado à mão, impossibilitando a prova formal de sua corretude, porém possiu uma codificação mais legível, caso queira-se estendê-lo. Também, o código de parsing não é gerado e o código pode ser otimizado à mão, tornando o código mais eficiente em alguns casos que um gerador de compiladores seria conservador, pelo fato de ser genérico.

Graças as características descritas, será utilizado o *frontend* para prototipagem pelo fato de seu código ser mais interessante que o do GCC e pela sua maior facilidade de trabalhar-se, ressaltada pelo paradigma de orientação à objetos (C++). Clang é um projeto voltado para ser estendido, porque compila C , C++, Objective-C e Objective-C++ num só frontend.

Outra vantagem ao usar o Clang é que as construções de C já estão prontas, portanto, precisamos somente implementar as instruções específicas da linguagem. É possível utilizar-se de todas as ferramentas que fazem análise, interpretação e modificação de código na infra-estrutura LLVM, inclusive interpretadores de sua linguagem intermediária (modo interpretado de execução).

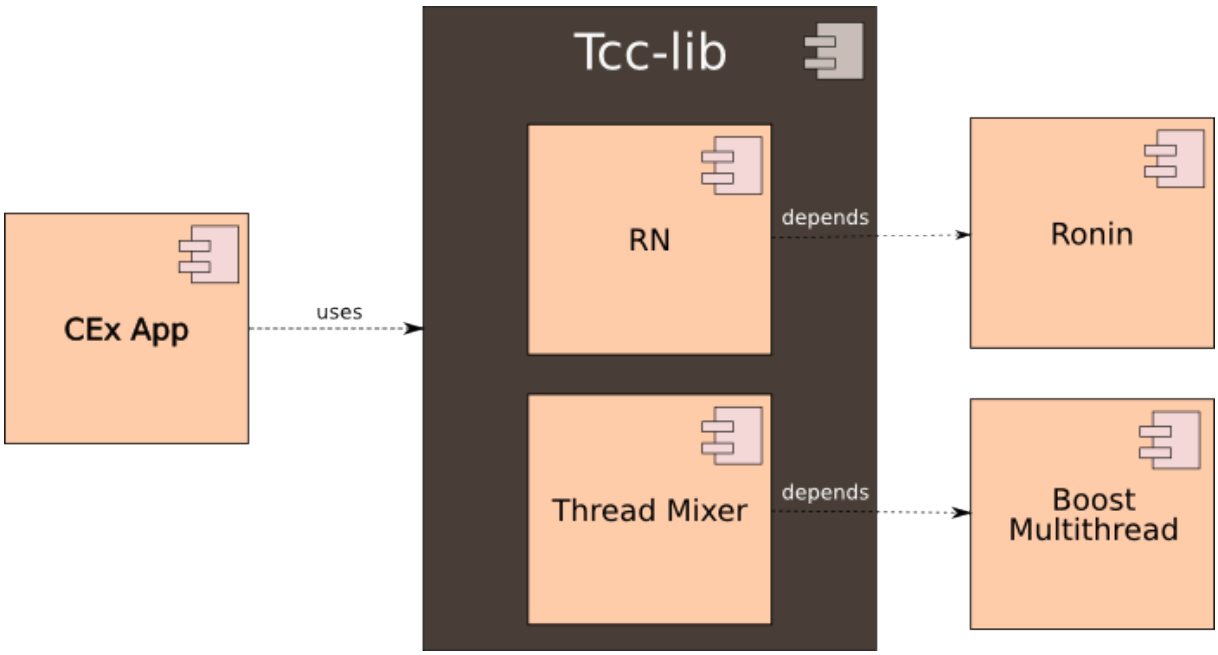

Figura 4.2: Arquitetura básica de um aplicativo CEx

A utilização do Clang é interessante de modo a aliviar grande parte da dificuldade de construir um compilador inteiro. Contudo, o Clang é para linguagens da família C e não está estável ainda para compilação de código estilo C++ (no qual ele é feito).

4.3 Biblioteca básica

Nessa seção são explicadas as peculiaridades mais interessantes relacionadas as bibliotecas que compõe a linguagem e ao trabalho final em si. Com o intuito de prover as facilidades específicas de paralelismo e primitivas para renderização gráfica, deve ser implementada uma biblioteca básica, suportada por algumas camadas de abstração, não visíveis diretamente para o programador. Dentre essas, destacam-se duas que são o motor gráfico Ronin e a Tcc-lib. Assim, a Tcc-lib fornece as funções de concorrência e tempo real, enquanto o motor gráfico Ronin providencia funções de renderização, colisão, etc. Uma lista das funções da biblioteca CEx pode ser observada nos anexos e um aplicativo CEx qualquer segue a arquitetura vista na figura 4.2.

4.3.1 Funções de paralelismo

As funções de paralelismo e concorrência necessitam de uma API específica para elas. Essas funções visam dar um tipo diferente de controle, complexo demais para ser adicionado

a sintaxe da linguagem (precedência, dependência entre Tasks, etc). Durante a graduação, em disciplinas relacionadas a programação concorrente, tivemos oportunidades de criar pequenos frameworks para facilitar a implementação de estruturas de concorrência. Um desses casos é a Tcc-lib, que foi levada além da parte de aula para possuir recursos avançados de gerenciamento de Threads.

A biblioteca Tcc-lib foi criada com o intuito de fornecer ferramentas voltadas para a programação concorrente, tais como *Monitor de Hoare*(Hoare 1974) e *Thread Mixer.* Serão fornecidas também abstrações de muitas funcionalidades oferecidas pelo próprio sistema operacional, como *mutex, semáforo, conditions, barreira e threads* utilizando-se da linguagem orientada a objetos de alto nível C++ sobre a API de multi-threading dos POSIX, denominada Pthreads(IBM 1998).

Uma *feature* interessante para a Tcc-lib seria evitar deadlocks simples por meio de *troca de prioridades*. Quando um processo de prioridade maior aguarda a liberação de um recurso relacionado a um processo de prioridade menor, essas prioridades são invertidas de modo a não haver um deadlock. Tal técnica pode ser observada em (Gehani 1991), na extensão da linguagem Concurrent C para problemas de tempo real.

O principal desafio será integrar os serviços fornecidos pela biblioteca aos serviços fornecidos pela engine Ronin e finalizar o *Thread Mixer* de modo a torná-lo compatível com todas as features da linguagem.

Thread Mixer

É a parte da biblioteca que permite execução de várias *Tasks*, relacionadas em *Jobs*, semelhante ao *Thread Weaver*(Boehm 2008) que é um gerenciador de threads , presente na kdelibs do KDE (interface gráfica multi-plataforma). O principal objetivo dessa camada é prover robustez e facilidade de paralelização de tarefas a serem executadas, abstraindo assim toda a parte de criação de Threads e utilização efetiva das unidades de processamento disponíveis.

4.3.2 Funções para primitivas gráficas

O suporte a renderização de primitivas gráficas e controle desses objetos primitivos faz parte das necessidades básicas de uma API de jogos, e para tanto, nenhuma solução já existente, individualmente, pareceu-nos completa e interessante o suficiente. Por isso e pelos motivos já descritos decidimos fazermos nossa própria engine.

A engine *Ronin*(Alves 2008), criada inicialmente por Felipe Borges Alves e codificada também por André Ferreira, tem suporte a programação de primitivas gráficas por meio do OpenGL (internamente).

Entre as características presentes na engine estão: Renderização por meio de objetos de alto nível, animações, texturas complexas/simples, detecção de colisão otimizada, view culling, carregamento de imagens em diversos formatos (.bmp, .png, .jpeg), carregamento de objetos complexos (.obj, .md2), sockets de alto nível, simulação física integrada com Bullet, a qual foi analisada em (Boeing e Bräunl 2007), carregamento de shaders, abstração de entrada (teclado, mouse, joystick, eventos de janela). De modo a entender como funciona internamente a API a ser proposta, descreve-se na próxima sessão a arquitetura interna do motor gráfico que possibilita as funcionalidades descritas. Um elemento faltante ainda é um editor de mapas para a engine. Uma solução seria utilizar algum formato emprestado de um outro editor por meio do carregamento do formato gerado por esse.

4.4 Engine Ronin

Nesta seção é apresentada a arquitetura da engine de jogos 3D Ronin, é explicado o funcionamento dos módulos de física, de detecção de colisão, de renderização e a arquitetura básica geral da *engine*.

4.4.1 Arquitetura básica

Os módulos básicos da engine são: Renderização, Física e Detecção de colisão. Para física é utilizada a biblioteca bullet(Physics Simulation Forum 2009), enquanto a renderização e a detecção de colisão[1] são implementados pela própria engine.

Como pode ser observado na figura 4.3, a cena e o módulo de detecção de colisão utilizam uma árvore de volumes hierárquicos regulares para armazenar os objetos. Ao renderizar uma cena , o módulo de renderização acessa a árvore, fazendo uso de suas propriedades para otimizar a renderização.

[1]Embora a física (bullet) tenha a sua própria detecção de colisão, ainda assim é necessário um segundo sistema de detecção de colisão, pois normalmente a física utiliza volumes aproximados, o que resulta em uma detecção de colisão imprecisa, já no módulo de detecção de colisão, existe a possibilidade de fazer teste de colisão a nível de triângulos, o que resulta em um teste mais preciso e compatível com o que é desenhado.

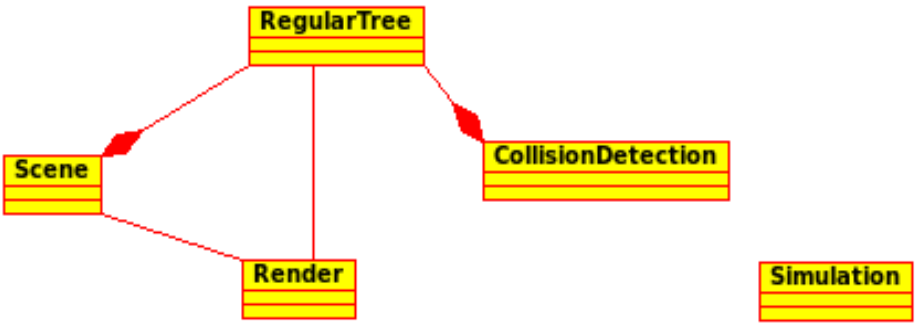

Figura 4.3: Arquitetura básica da engine

4.4.2 Árvores de volumes hierárquicos

Para otimizações de renderização e usos diversos, a engine utiliza-se de várias implementações de árvores de volumes hierárquicos(Akenine-Möller, Haines e Hoffman 2008), sendo que cada implementação tem é usado de forma ligeiramente diferente. A seguir é discutido cada uma das implementações, bem como suas propriedades e onde é feito o seu uso.

Regulares

Na engine são implementados dois tipos de árvores de volumes hierárquicos regulares, as *octrees* e as *quadtrees*. As *octrees* são semelhantes as *quadtrees*, divergindo apenas no fato das octrees se subdividirem em um eixo a mais. Por conveniência aqui será explicado apenas o funcionamento da quadtree, uma vez que conhecendo o mesmo é fácil expandir os conceitos para uma dimensão a mais.

As *quadtrees* consistem na subdivisão recursiva de cada nodo em quatro subnodos. Os subnodos são o resultado do nodo pai cortado ao meio nos eixos[2] X e Y. Um problema das quadtrees são os objetos nas divisões dos nodos. Caso um objeto esteja no centro do nodo, este não caberá completamente em nenhum dos nodos filhos, um objeto no centro da quadtree ficará no nodo raiz, o que terá um impacto negativo na performance. Uma solução para este problema é o uso de *quadtrees/octrees* com folga(Eppstein, Goodrich e Sun 2005) (figura 4.4).

Binary Space Partitioning Tree

Binary Space Partitioning Trees(Akenine-Möller, Haines e Hoffman 2008) ou simplesmente *BSPTrees* são árvores de volumes hierárquicos onde cada nodo é subdividido em dois nodos

[2]Na engine são utilizados os eixos X e Z.

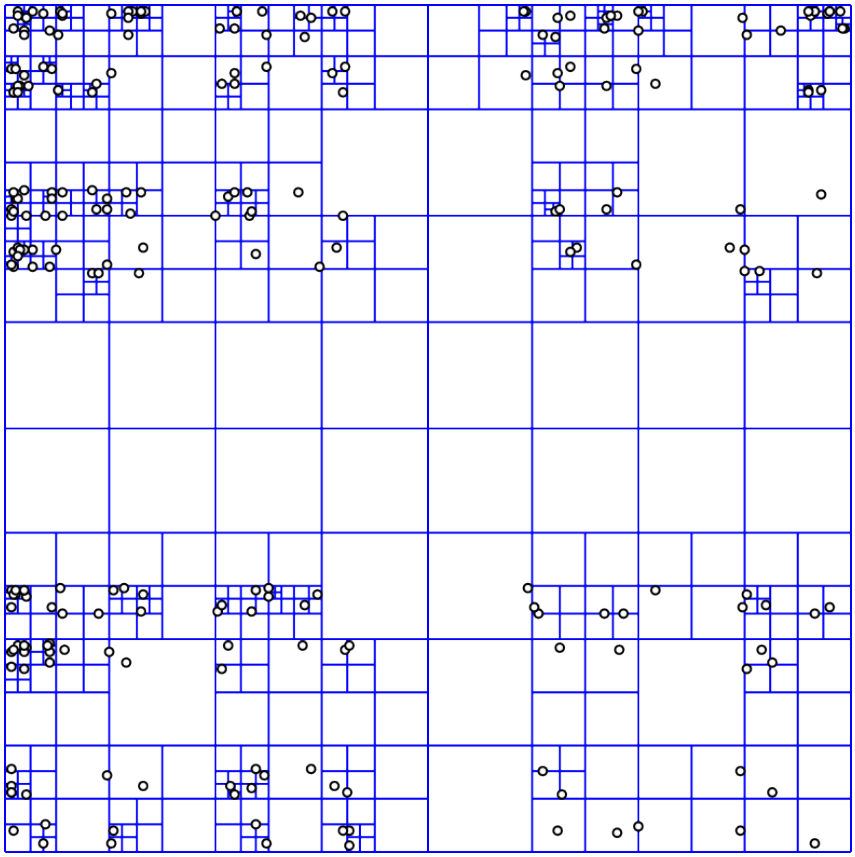

Figura 4.4: Exemplo de uma quadtree, como vista em (Eppstein, Goodrich e Sun 2005)

menores.

Ao contrário das octrees/quadtrees, as subdivisões não são regulares, o que normalmente resulta em uma árvore mais equilibrada, isto é, os dois nodos da subdivisão ficam com uma quantidade mais próxima de objetos, entretanto esta estrutura não é adequada para ser constantemente modificada. Devido a esta propriedade as árvores regulares são utilizadas para manter os objetos, enquanto as *BSPTrees* são utilizadas para detecção de colisão a nível de triângulos, sendo calculada apenas uma vez e depois não mais modificada.

4.4.3 Renderização

O módulo de renderização consiste em uma classe abstrata (Render) permitindo várias implementações de renderizadores, cada um com características diferentes. As opções de módulos de renderização atualmente são: DebugRender (renderização para depuração), SingleThreadRender (renderização realizada utilizando apenas uma thread) e MultiThreadRender (renderização realizada de forma a melhor aproveitar o processamento de um computador multi-core), cada uma dessas implementações será discutida em detalhes mais a frente.

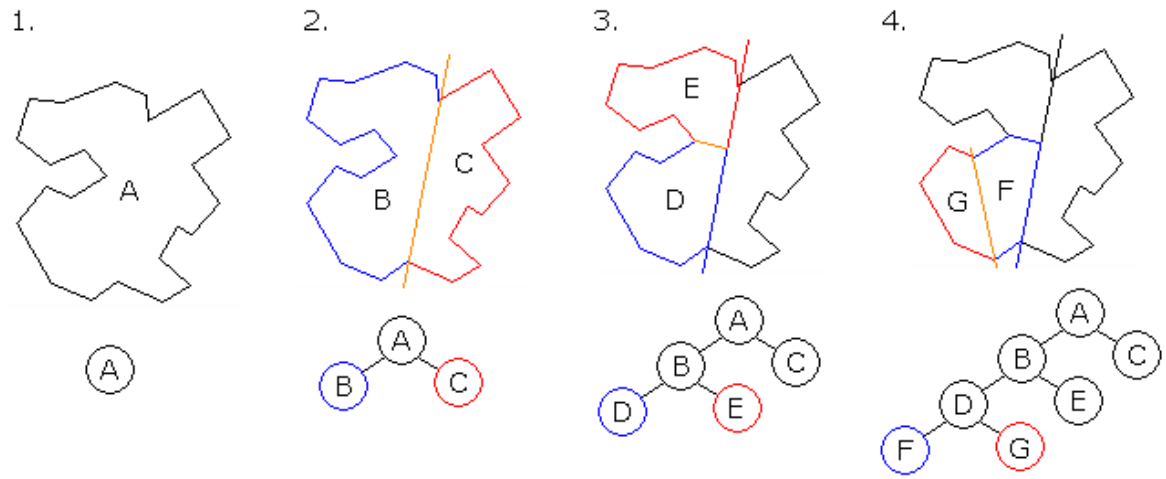

Figura 4.5: Construção de uma BSP Tree

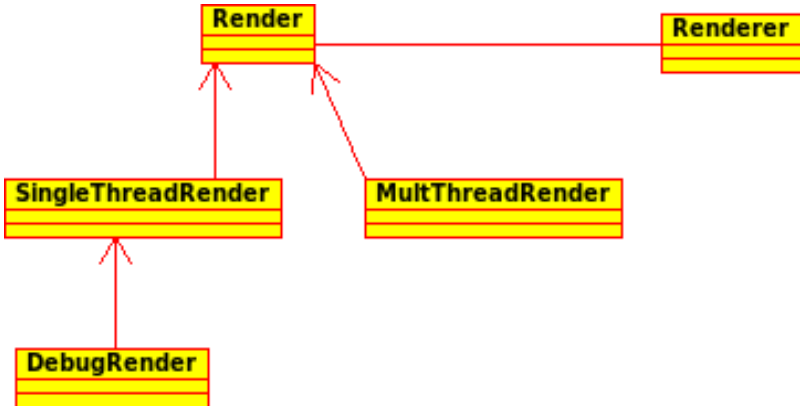

Figura 4.6: Classes responsáveis pela renderização

O módulo de renderização em si não faz desenho algum, o mesmo apenas gerência os renderizadores, os quais são responsáveis apenas pela renderização de um único objeto ou efeito visual.

Viewclipping

Dentre as otimizações na renderização implementada pela engine, podemos destacar o *viewclipping*. O *viewclipping* consiste em não desenhar objetos que estão fora da região visível. Um problema causado pelo uso do *viewclipping*, é que torna-se necessário testar se um determinado objeto está na região visível, e caso a maioria dos objetos estejam visíveis, o desempenho pode acabar sendo na verdade prejudicado.

Enquanto testar a visibilidade de cada triângulo do objeto nos diz com precisão se o objeto está ou não na região visível, este teste muito provavelmente levaria mais tempo para ser

realizado do que simplesmente desenhar o objeto incondicionalmente. Para minimizar o processamento utilizado para determinar se um objeto está ou não visível, uma primeira simplificação é não fazer um teste exato, em outras palavras, ao invés de testar cada primitiva do objeto, calcula-se apenas uma caixa que contém o objeto inteiro (*bounding box*) e realiza-se o teste apenas com esta caixa. Caso seja determinado que a caixa não está visível não é necessário desenhar o objeto. Note que não há problema em desenhar um objeto que não está visível, neste caso apenas gasta-se tempo a mais, porém deixar de desenhar um objeto visível é considerado um erro.

Para otimizar ainda mais, minimizando os testes de intersecções de *bounding box*, utiliza-se uma Árvore de volumes hierárquicos regular. Inicia-se na raiz da árvore, é realizado um teste com cada subnodo da árvore, caso o subnodo não esteja visível, pode-se descartar todo aquele ramo da árvore, caso determine-se que o subnodo está completamente dentro da região visível, tudo que está abaixo do mesmo é desenhado sem a necessidade de realizar mais testes, caso o nodo esteja parcialmente visível os objetos neste são testados (e possivelmente desenhados), e é repetido o teste para seus subnodos recursivamente.

SingleThreadRender

A classe SingleThreadRender implementa a renderização em uma única thread. A utilidade desta classe se da nos casos onde o desempenho dela é melhor que a renderização multithread que quando não há testes de viewclipping o bastante a paralelizar para compensar o overhead causado pela paralelização. O MultiThreadRender implementa apenas a renderização utilizando viewclipping, caso sabidamente todos, ou a maioria, dos objetos estão visíveis, e consequentemente não é vantajoso utilizar viewclipping, o uso da classe SingleThreadRender é recomendado.

DebugRender

A classe DebugRender tem o objetivo de facilitar a depuração tanto da própria engine como dos jogos nela sendo desenvolvidos. Este módulo de renderização permite que sejam desenhadas informações úteis para depuração como por exemplo a *bouding box* dos objetos. A classe DebugRender herda de SingleThreadRender reaproveitando-se assim o algoritmo de renderização, o que permite que a classe DebugRender preocupe-se apenas com as informações visuais pertinentes a depuração.

MultiThreadRender

A renderização em múltiplas threads tem como objetivo máximizar a utilização do processamento de computadores multi-core. Uma vez que a API do OpenGL não é thread-safe, isto é, não é possível desenhar efetivamente dois objetos simultaneamente, a renderização em múltiplas-threads foca em otimizações, como viewclipping, de forma a paralelizar o teste de intersecção com o frustum (o que pode ser feito em N threads) e a renderização dos objetos (o que necessariamente é feito em apenas uma thread).

Foi implementado um pipeline de produtor e consumidor com os objetos a serem desenhados, foram feitas várias otimizações de forma a evitar que a thread principal (a que está desenhando os objetos) precise esperar por objetos para desenhar, uma vez que a tarefa executada por esta thread é o gargalo da renderização.

Um dos problemas com a renderização em múltiplas threads é que nem sempre são necessários testes o suficiente pare que o ganho com a paralelização supere o overhead inserido pela mesma, deixando as threads responsáveis pelos testes de intersecção a maior parte do tempo IDLE e consequentemente não aproveitando todo o poder de processamento disponível. Nas situações onde isso ocorre é recomendado a utilização da renderização sequencial.

4.4.4 Detecção de colisão

A detecção de colisão possui um foco diferente da implementada na física. Ao invés de detectar colisões de todos os objetos simultaneamente, visando a reação das colisões, o módulo de detecção de colisão detecta todas as colisões de um objeto específico, ou ainda, permite que seja utilizado um objeto geométrico arbitrário para fazer o teste.

O módulo utiliza uma árvore de volumes hierárquicos regulares para organizar os objetos no espaço. Devido a utilização da árvore, a busca por objetos cuja *bounding box* intersecta a geometria do teste se torna eficiente. Uma vez encontrados os objetos cujas bounding boxes intersectam a geometria, é feito um teste mais preciso. Para a realização do segundo teste, é utilizado um algoritmo específico de cada objeto, definido na sua criação. Os algoritmos disponíveis são:

- Colisão simples - É realizado apenas um simples teste do objeto em questão com um volume (esfera ou cubo).
- Lista de volumes - É realizado um teste com cada volume (esfera ou cubo) em uma lista. Se um dos volumes colide, a colisão é marcada.

- Árvore de triângulos - É utilizada uma *BSPTree* contendo triângulos, se um dos triângulos da árvore colidir com o objeto em questão, a colisão é marcada.

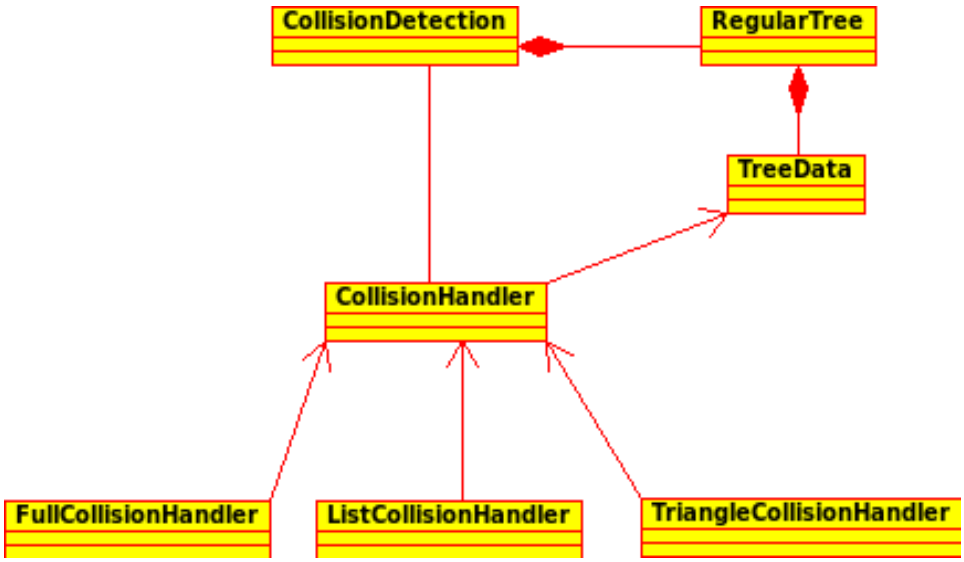

Figura 4.7: Módulo de detecção de colisão

4.4.5 Física

Para implementar o módulo de física é utilizada a biblioteca Bullet(Physics Simulation Forum 2009). O módulo de física implementa o padrão de projetos fachada, proposto em (Gamma et al. 1994), para facilitar o uso da física e integração dos objetos da simulação com o resto da engine.

4.4.6 Demais módulos

Dentre os outros módulos importantes, porém não demonstrados no diagrama da arquitetura básica (figura 4.3), estão os módulos de *input* (entrada) e carregamento de *shaders*. O módulo de *input* trata-se do suporte para eventos básicos da engine, como eventos de *mouse*, teclado, *joystick* e os eventos relacionados a gerenciamento de janelas (perda de foco, redimensionamento). Para tal, ele utiliza-se do padrão de projeto observador, como visto em (Gamma et al. 1994). Já o carregamento de shader, dá-se pelo padrão fábrica abstrata, derivado do padrão fábrica, possibilitando interfaces de fábricas para diferentes linguagens de *shading*, definido em (Gamma et al. 1994).

Algoritmo 4.1: Exemplo de código LLVM gerado para o algoritmo 3.2

```
void main() {
...
    synchronized (produceCondition, arrayMutex) {//synchronized before
==============
synchronized.before:          ; preds = %while.body
    %0 = load %struct.synchronizer* @produceCondition ; <%struct.
        synchronizer> [#uses=1]
    call void @__builtin_presync(%struct.synchronizer %0)
    %1 = load %struct.synchronizer* @arrayMutex ; <%struct.synchronizer> [#
        uses=1]
    call void @__builtin_presync(%struct.synchronizer %1)
    br label %synchronized.body
==============

        while (1) ... //synchronized body
==============
    synchronized.body:        ; preds = %synchronized.before
    br label %while.cond1
    ...
==============

    }//synchronized after
==============
synchronized.after:
==============
```

4.5 Implementação das instruções

Na implementação das instruções propostas no capítulo anterior, precisou-se adotar estratégias diferentes de implementação para cada uma das instruções (ou conjunto delas).

De modo a tornar o mais genérico possível a implementação adotada, decidiu-se que as instruções seriam implementadas por meio de uma biblioteca intermediária, a qual desprenderia a funcionalidade alvo de implementação no compilador. A seguir, encontram-se as explicações das soluções utilizadas (obs:. os códigos gerados são não-otimizados).

4.5.1 Instruções do_rt e synchronized

Para o do_rt e synchronized foram adotadas as mesmas estratégias, portanto eles são agrupados como grupo de solução nessa seção. No código llvm gerado, cria-se um prólogo e epílogo no bloco alvo do do_rt/synchronized. Em ambos, adiciona-se uma chamada para uma função com uma assinatura do tipo "__builtin_NomeDaKeyword_pre" e "__builtin_NomeDaKeyword_pos", para o prólogo e o epílogo (nessa sequência). No prólogo do do_rt, passam-se a expressão referente a *deadline*, enquanto no synchronized são passadas as variáveis utilizadas. No epílogo

Algoritmo 4.2: Código LLVM para parallel no algoritmo 3.2

```
void main() {
...
    else
        parallel {// parallel before
==================
define void @__builtin_parallel1_main.c() { ;nome da função forjada
entry:
==================

        printf("Current sum %d\n", sum); // parallel body
==================
    %tmp = load i32* @sum        ; <i32> [#uses=1]
    %call = call i32 (i8*, ...)* @printf(i8* getelementptr ([16 x i8]* @.
        str, i32 0, i32 0), i32 %tmp)       ; <i32> [#uses=0]
    ret void
==================

    }//parallel after
==================
}
==================
```

do do_rt passam-se o identificador da deadline, retornado pela função chamada no prólogo, enquanto no synchronized apenas as variáveis são passadas (na mesma sequência) para a função de liberação dos mecanismos de sincronia. A geração de código para a instrução synchronized do algoritmo 3.2 é demonstrada no 4.1.

4.5.2 Instruções async, periodic_rt e parallel

Na implementação do async, periodic_rt e parallel foram adotadas o mesmo tipo de estratégia. Os blocos referentes a qualquer um dessas instruções são removidos do corpo da função e uma função especial é forjada (em tempo de compilação), acomodando o código contido. No lugar do corpo da função fica uma chamada para a biblioteca intermediária do tipo "__builtin_NomeDaKeyword_call", passando um ponteiro para a função forjada como parâmetro e os outros parâmetros específicos de cada instrução. O parallel_for também se encaixa nessa mesma categoria, mas sua estrutura especial será explicada na próximo seção. Um exemplo de geração de código desse tipo de instrução pode ser observado no algoritmo 4.2.

4.5.3 Instrução parallel_for

Para implementar o parallel_for, também foi adotada a estratégia de inserir uma chamada no corpo da repetição e forjar uma função que é passada como parâmetro para a biblioteca.

Algoritmo 4.3: Código intermediário LLVM para o algoritmo 4.7

```
void main() {
    parallel_for(int i = 0; i < 40; ++i) {
==================
    %0 = call %struct.job @TM_createJob() ; cria o job
for.cond:
    ...
    br i1 %cmp9, label %for.body10, label %for.end...
==================

        ...
==================
for.body:
    ...
    call void @__builtin_parallel_for_call(void (i32)* @__builtin_parallel_for0_main.c,
        %struct.job %0, i32 %1) ;cria task para a condição a iteração i

==================

    }
==================
for.end:
    call void @TM_execAndWaitJob(%struct.job %0)
    ...
==================
}

==================
define void @__builtin_parallel_for0_main.c(i32 %i) { ;função forjada
entry:
    ... ; corpo do for original
}
==================
```

Computador	Dual Core	Quad Core
Memória	4GB - 800mhz	4GB - 800mhz
Placa de vídeo	Geforce 9600M GT 512MB	Geforce 9600 GT 512 MB
Processador	Intel Core 2 Duo 2.13Ghz	AMD Phenom X4 2.0Ghz

Tabela 4.1: Configurações

Porém, dada a complexidade maior, graças a estrutura ser iterativa, insere-se uma chamada que cria uma estrutura de dados que será utilizada para armazenar as chamadas de cada iteração no prólogo do parallel_for. Para preencher essa estrutura, itera-se normalmente no *loop*, passando ela para a camada intermediária na chamada de função e por fim, ao acabar o *loop*, manda-se executar todas as tarefas encontradas. Naturalmente, a expressão de condição de iteração não deve possuir dependência de execução com o corpo da estrura de repetição.

Um exemplo de geração de código para o parallel_for pode ser observado no algoritmo 4.3. Para demonstrar a utilização e eficiência das construções básicas da linguagem, será feita uma demonstração de exemplos clássicos e interessantes na próxima seção.

4.6 Problemas clássicos e comparativos

Nessa seção, será demonstrada a implementação de alguns problemas clássicos com as instruções da linguagem e sua biblioteca. Entre os problemas clássicos demonstrados estão o de solução de fibonacci recursivo (sequencial e paralelizado), o jantar dos filósofos e a visualização de fractais. Para os testes foram utilizados dois ambientes de simulação os quais podem ser observados na tabela 4.1.

4.6.1 O problema do jantar dos filósofos

O enunciado do problema do jantar dos filósofos foi proposto primeiramente por Tony Hoare, baseado em Dijkstra(Dijkstra 1965). O problema trata de 5 filósofos que estão reunidos para um jantar. Para comer, os filósofos dispõe de um prato para cada e 5 garfos intercalados entre seus pratos. As premissas quanto ao problema do jantar são as seguintes:

- Os filósofos não conversam uns com os outros (não há comunicação entre threads)
- Para apreciar o macarrão, os filósofos precisam utilizar dois garfos
- Os filósofos ou estão comendo ou estão pensando
- Eles só podem utilizar os garfos adjacentes a seus pratos

O problema do jantar dos filósofos trata-se de um problema de exclusão mútua, onde sua solução pode ser estendida para muitos outros problemas relacionados à programação concorrente. Uma possível solução escrita em CEx para o problema encontra-se no algoritmo 4.4 (sem *livelocks*).

4.6.2 Sequência de Fibonacci

A sequência de Fibonacci foi assim chamada, após Leonardo de Pisa (conhecido como Fibonacci - "filho de Bonaccio") tê-la apresentada em seu trabalho. Originalmente a sequência foi proposta na Índia. Sua definição é a seguinte:

$F(0) = 0$

$F(1) = 1$

$F(n) = F(n-1) + F(n-2)$

Dessa forma, a sequência seria $\{0,1,1,2,3,5,8...\}$. Para calcular essa sequência pode-se fazer um algoritmo recursivo, como visto no algoritmo 4.5.

Algoritmo 4.4: Jantar dos filósofos modelado com a linguagem. Versão sem *livelocks*

```
#include <inst/parallel.h>
#include <sync/sync.h>

#include <printf.h>
#include <stdio.h>
#include <stdlib.h>

const unsigned int FORK_NUM = 5;
const unsigned int PHILOSOPHER_NUM = 5;
const unsigned int NUM_OF_ITERATIONS = 30;/*would loop forever with no limit*/
/*One mutex per fork, one for id synchronization and a barrier (main thread won't exit)*/
synchronizer forkSync[FORK_NUM], idSync, barrier;
int id = 0;/*Philosopher id -- auto-incremented*/

void init() {
    job j;
    for (int i = 0; i < FORK_NUM; ++i)
        forkSync[i] = TM_createMutex();
    idSync = TM_createMutex();
    barrier = TM_createBarrier(PHILOSOPHER_NUM);
}

void eat(int id) {
    printf("Philosopher %d: eating\n", id);
    sleep(1);
}

void think(int id) {
    printf("Philosopher %d: Thinking\n", id);
    sleep(1);
}

void createPhilosopherAndExec() {
    int myId, leftForkIndex, rightForkIndex;

    synchronized(idSync) { myId = ++id; }

    leftForkIndex = (myId + FORK_NUM -1) % FORK_NUM;
    rightForkIndex = myId % FORK_NUM;
    synchronizer leftForkSync = forkSync[leftForkIndex],
        rightForkSync = forkSync[rightForkIndex];

    if (myId != PHILOSOPHER_NUM) {
        for (int i = 0; i < NUM_OF_ITERATIONS; ++i) {
            synchronized (leftForkSync, rightForkSync) {
                eat(myId);
            }

            think(myId);
        }
    }
    else {
        for (int i = 0; i < NUM_OF_ITERATIONS; ++i) {
            synchronized (rightForkSync, leftForkSync) {
                eat(myId);
            }

            think(myId);
        }
    }
    TM_waitBarrier(barrier);
}

void main() {
    init();
    printf("Beginning Philosophers Dinner\n");
    for(int i = 0; i < PHILOSOPHER_NUM; ++i)
        async (LONG_JOB) { createPhilosopherAndExec(); }
    TM_waitBarrier(barrier);
    TM_destroyBarrier(barrier);
}
```

Algoritmo 4.5: Algoritmo recursivo simples para cálculo do número de Fibonacci $F(n)$

```
int fibo(int n) {
    if (n <= 1) return n;
    else return fibo(n-1) + fibo(n-2);
}
```

Algoritmo 4.6: Cálculo iterativo de números de Fibonacci utilizando-se da versão simples (4.5)

```
void main() {
    for(int i = 0; i < 40; ++i) {
        printf("fibo(%d): %d\n", i, fibo(i));
    }
}
```

Fibonacci simples sequencial vs paralelo

Para fins de comparação, efetua-se um loop iterativo calculando números de fibonacci, utilizando-se de laços tradicionais (algoritmo 4.6) e o laço paralelizado (algoritmo 4.7). A tabela 4.2 apresenta os comparativos obtidos com vários níveis de otimização para cada computador. Os tempos ali observados são exclusivamente entre o começo e o término das iterações e os resultados são a média de quatro execuções.

Os tempos observados são esperados para as iterações paralelizadas, porém os resultados quanto à uma thread em múltiplos núcleos são mais rápidos que em relação ao for comum. Isso deve-se ao fato da condição do for estar rodando em paralelo ao corpo (a Thread principal calcula a condição e as outras as iterações). Também, várias funções da biblioteca são usadas de forma assíncrona, causando os núcleos extras a serem utilizados mesmo num código sequencial. Outro ponto interessante, é que para cada ciclo de execução do parallel_for, na implementação feita, espera-se o laço terminar (executado quatro vezes), diminuindo o *throughput* (vazão) de execução de tasks. Essa funcionalidade pode ser configurada dinamicamente (por chamada da biblioteca).

Nos tempos relacionados à execução em um Quad Core fica claro o ganho linear em relação

Algoritmo 4.7: Cálculo paralelo de números de Fibonacci utilizando-se da versão simples (4.5)

```
void main() {
    parallel_for(int i = 0; i < 40; ++i) {
        printf("fibo(%d): %d\n", i, fibo(i));
    }
}
```

Tipo - Threads / PC	*Dual Core*	*Quad Core*
for - X	4.36	3.26
parallel - 1	4.29	2.98
parallel - 3	2.62	1.54
parallel - 6	2.58	1.35

Tabela 4.2: Tempos, em segundos, de iteração para otimização O3

ao número de threads utilizadas e também pode-se observar um certo número ótimo de threads em relação ao número de processadores, que foi descoberto ser algo em torno de 1,5 vezes o número de processadores[3].

4.6.3 Visualização de fractal

Um fractal é uma forma geométrica única ou fragmentada que pode ser dividida em partes, cada uma representando uma cópia do todo. Essa propriedade, inerente dos fractais é chamada de auto-similaridade.

Os fractais foram definidos com esse nome por Benoît Mandelbrot, cujo nome foi derivado do latim "*fractus*", que significa "quebrado" ou "fraturado".

Propriedades comuns aos fractais:

- Definição simples e recursiva
- Incapaz de ser analisado pela geometria euclidiana
- É auto-similar
- Seu conjunto de pontos faz parte do plano complexo

Dentre os fractais mais conhecidos encontram-se os de Mandelbrot e os de Julia, cuja correlação do tipo Hausdorff de duas dimensões (mapeamento), pode ser observado em (Shishikura 1991). A seguir demonstra-se uma implementação do conjunto de Mandelbrot utilizando a CEx.

Conjunto de Mandelbrot e conjunto de Julia

O conjunto de Mandelbrot é um conjunto de pontos definidos no plano complexo, e suas "bordas" formam um interessante estudo de caso de fractal. Sua definição matemática é simples e direta:

[3] Valor observado e não algo estudado extensivamente e provado formalmente

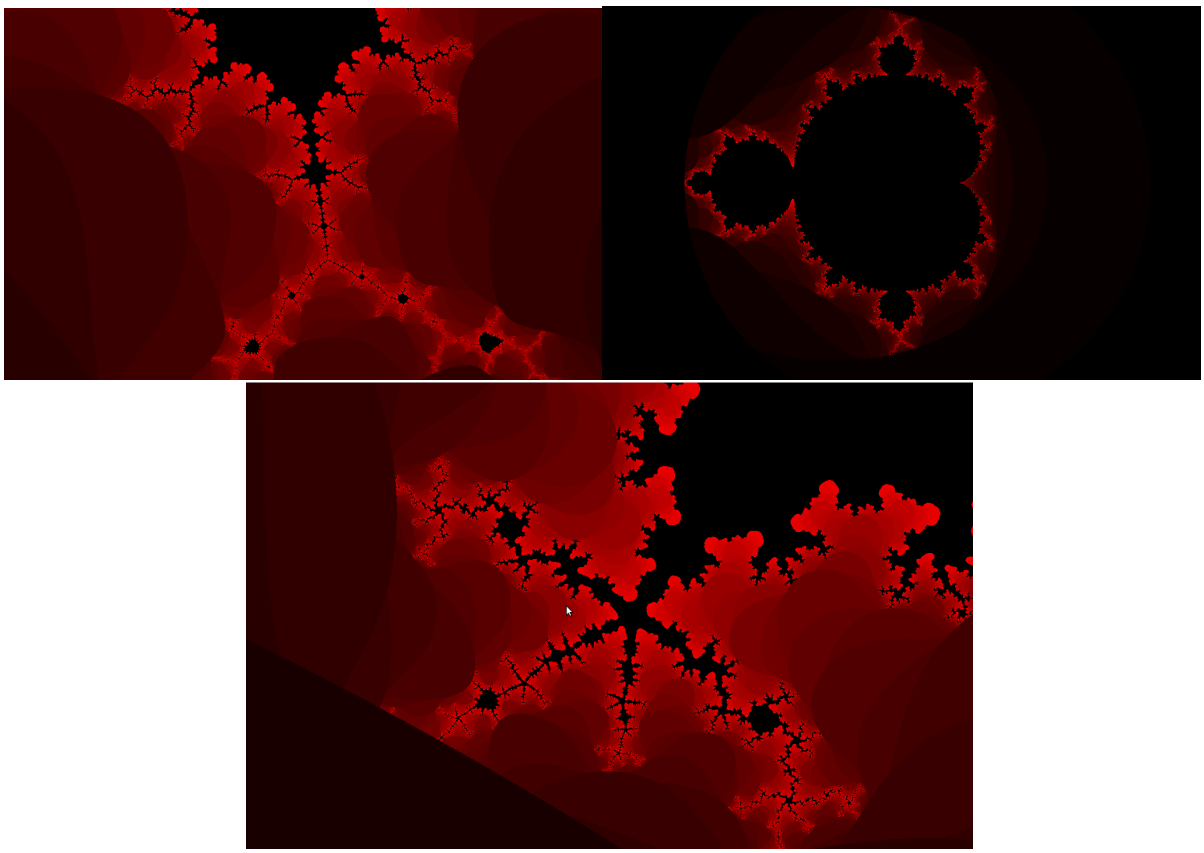

Figura 4.8: Figuras demonstrando diferentes pontos de um fractal do conjunto de Mandelbrot

PC / Tipo de laço	Dual Core (qps)	Quad Core (qps)
Laços sequenciais	9.5	7.43
Laços paralelizados	18.12	19

Tabela 4.3: Performance, em quadros por segundo (qps), das implementações

$$P_c : \mathbb{C} \rightarrow \mathbb{C}$$

$$P_c : Z_{n+1} \rightarrow Z_n^2 + c$$

$$M = \{c \in \mathbb{C} : \exists s \in \mathbb{N}, |P_c^n(0)| \leq s\}$$

Como simplificação, adota-se $s = 2$ e verifica-se em função disso se o número está convergindo ou não para s. Matematicamente, teria que fazer-se infinitas vezes até verificar se converge, mas em um programa de computador, simplifica-se delimitando um número máximo de iterações para verificar a convergência.

A implementação do conjunto de Mandelbrot em foram feitas em duas versões. Uma utilizando laços paralelizados e outra não. Os resultados visuais podem ser observados na figura 4.8, enquanto o benchmark comparativo de paralelização pode ser observado na tabela 4.3, para um tamanho de imagem de 1280 por 800 pixels.

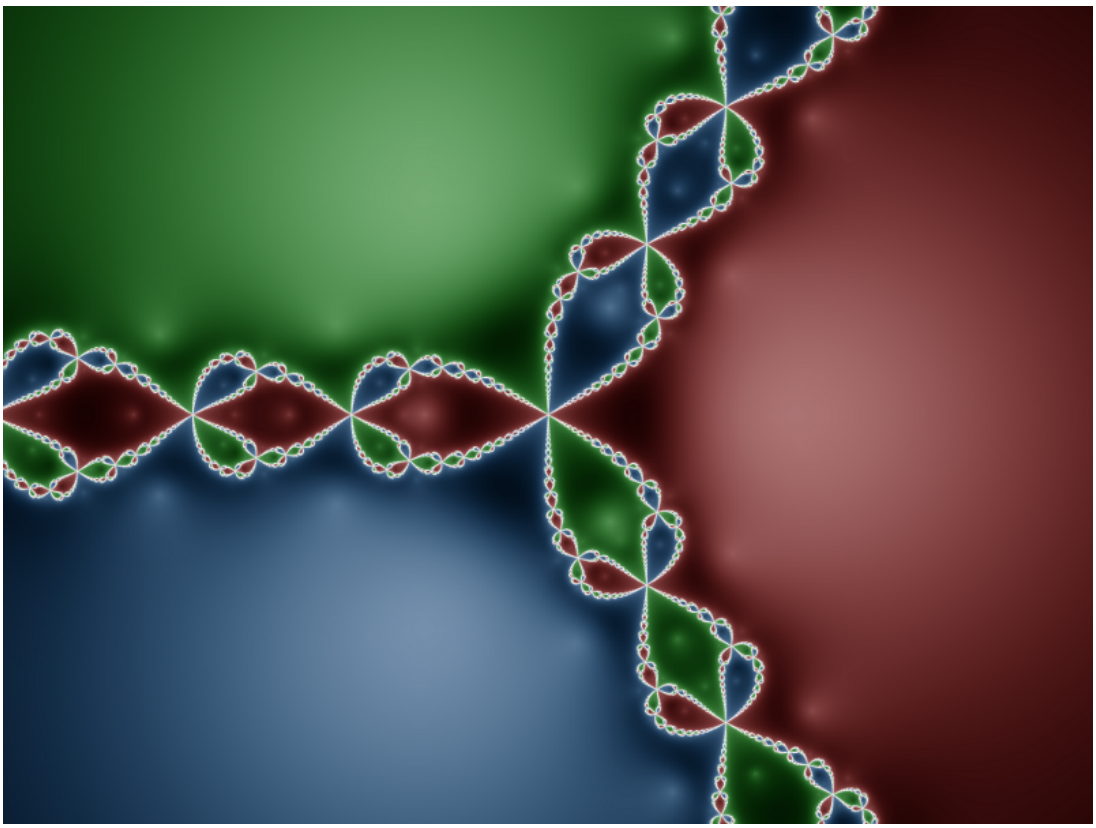

Figura 4.9: Exemplo de renderização de conjunto de Julia

O conjunto de Julia, é um subconjunto do conjunto de Mandelbrot, isto é, o conjunto de Mandelbrot é formado por infinitos conjuntos de Julia, uma vez que sua definição é a seguinte:

$P_c : \mathbb{C} \to \mathbb{C}$

$P_c : Z_{n+1} \to Z_n^2 + K$

$M = \{c \in \mathbb{C} : \exists s \in \mathbb{N}, |P_c^n(0)| \leq s\}$

K é uma constante complexa, fazendo do conjunto de Julia um subconjunto de Mandelbrot, com c = K. Um exemplo de um conjunto de Julia relacionado ao método de newton para $f : z \to z^3 - 1$ (parte branca) pode ser observado na figura 4.9 (as partes coloridas são relativas as raízes de f).

Renderização dependente de carga

Uma utilização interessante para o a instrução do_rt, é o controle de renderização dependente de carga que descarta os frames que demoram muito a serem desenhados. Para demonstração do funcionamento, em um conjunto de Mandelbrot, renderizou-se mesmo os quadros falhos, de modo a obter-se imagens incompletas e misturadas com as imagens anteriores ao

Algoritmo 4.8: Trecho de código de mandelbrot com renderização dependente de carga

```
void dropFrameCb(int signal,
        siginfo_t* siginfo, void* uc) {

    /* drops all simple jobs like the parallel_for ones */
    TM_clearSimpleJobs();
}

void doMandelbrotRendering() {
    ...
    TM_setRtSigCb(signal, dropFrameCb);
    ...
    do_rt (1.0 / AVERAGE_FPS)
        drawMandelbrotSet();
}
```

navegar-se pelos conjuntos. Tal efeito pode ser observado na figura 4.10. Para obter-se esse feito, apenas descarta-se os frames que estejam abaixo da média esperada de renderização, demonstrada na tabela 4.3. Abstratamente, tem-se o código demonstrado no algoritmo 4.8.

No exemplo demonstrado, a renderização paralelizada ocorre por uma linha inteira da imagem, fazendo com que quando captura-se a exceção de estouro de temporizador, os trabalhos que estão executando terminem de desenhar até o fim. Portanto, os "fragmentos" renderizados são uniformes, como pode-se observar.

Uma outra utilização poderia não só ser usada para descartar frames, como também para fazer algum tipo de otimização nos elementos a serem desenhados. Assim, é possível diminuir o tempo de renderização quando o computador não conseguir manter o ritmo de desenho por algum fator externo ou mesmo por ser incapaz de renderizar os elementos visíveis atuais no tempo desejado. Na próxima seção será demonstrado um exemplo de protótipo portado para CEx que originalmente foi escrito em C++.

4.7 Protótipo de física portado

A *engine* Ronin possui inúmeros testes que validam seus módulos. Para o módulo de física não é diferente e ele possui um exemplo que utiliza-se das mais variadas funções de simulação física.

O teste possui as seguintes características:

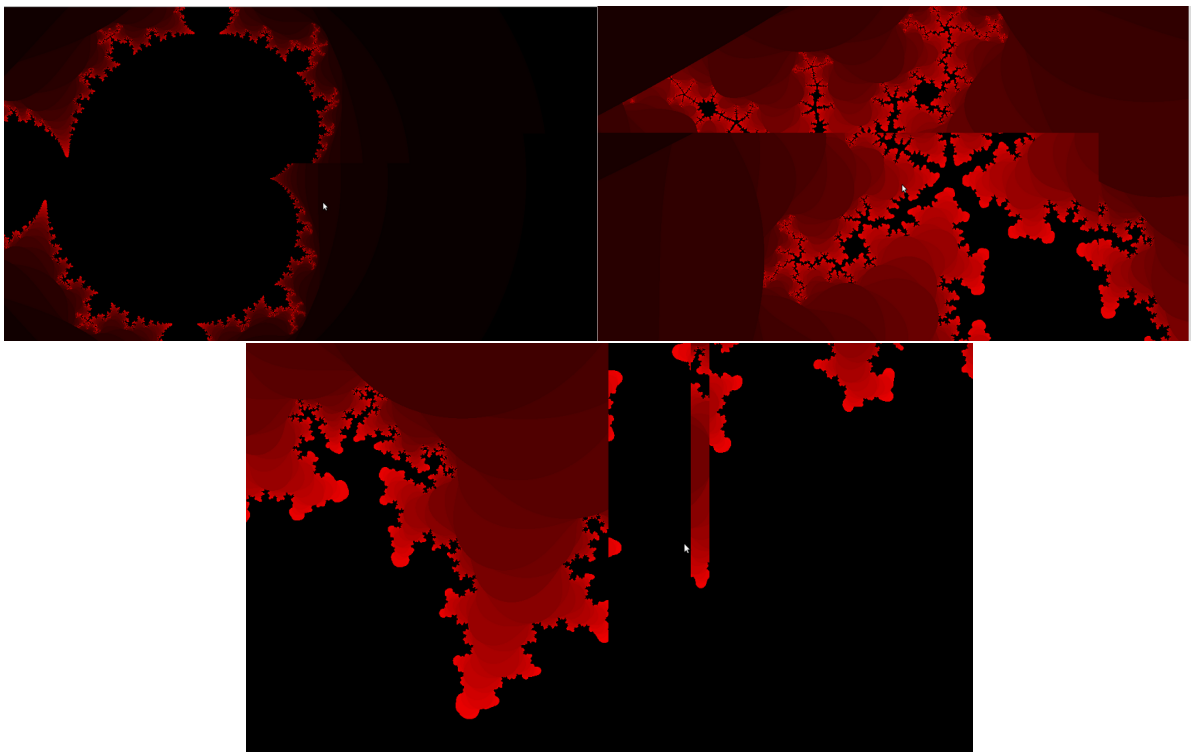

Figura 4.10: Renderizações incompletas de diferentes pontos da série

- Um paralelepípedo de massa infinita e não sofre ação de forças externas (como gravidade)
- Um conjunto de cubos que começam flutuando e sofrem a ação de uma força externa
- Tanto os cubos quanto o paralelepípedo são corpos rígidos
- É possível aplicar uma força com direção ao centro do paralelepípedo em todos os cubos e vetor resultante "para cima"

Para que fosse possível portar-se o código exemplo, escrito em C++, com a *engine* Ronin, inúmeras partes dela precisaram ser portadas para uma interface em C que abstraísse a orientação à objetos contida por baixo. Desse modo implementou-se o módulo "RN" (interface C para Ronin) que possui as abstrações que permitem mapear as abstrações de C++ para uma interface C equivalente.

Na figura 4.11, observa-se os seguintes passos: ao inicializar-se o sistema os cubos caem na direção do paralelepípedo pela força de atratividade da gravidade (I). Então, os cubos repousam, uns sobre os outros, na superfície do paralelepípedo e sua energia cinética nesse momento torna-se zero (II). Aplica-se uma força com direção ao centro do paralelepípedo e a força resultante é "para cima" após a colisão perfeitamente inelástica dos cubos (III). A força aplicada cessa e os cubos colidem e caem com vetores forças resultantes diferentes (IV). Os cubos então repousam sobre a superfície ou caem indefinidamente, caso ultrapassem a borda da superfície (V).

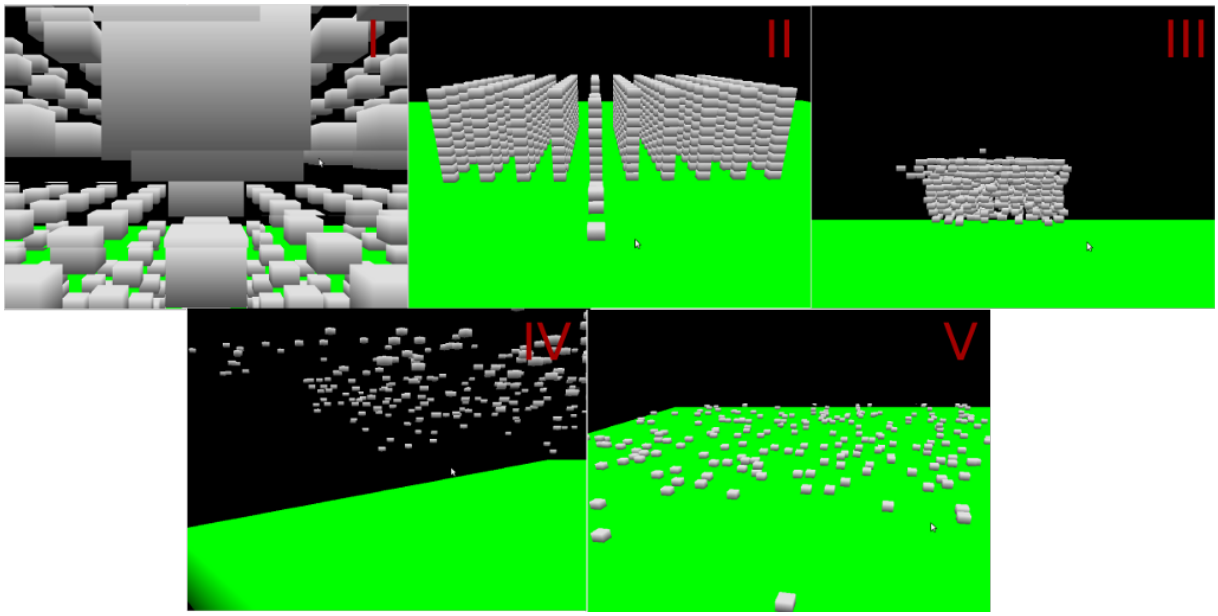

Figura 4.11: Ilustração demonstrando a execução do protótipo de física escrito em CEx

4.8 First person shooter

O intuito de codificar-se um FPS como protótipo de jogo é demonstrar a utilização geral das instruções da engine junto as bibliotecas criadas e validar as implementações feitas. Ele é um bom exemplo porque testa os módulos de renderização, entrada/saída, detecção de colisão, física num só protótipo.

Uma grande importância do FPS foi a finalização da Ronin C Interface (RN), a qual pode ser observada nos anexos. Essa API, provê uma interface estruturada para o código orientado à objetos da engine Ronin, sendo ela um trabalho extensivo de mapeamento de objetos em C++ para objetos em C.

Outro fator interessante que pôde ser feito com o jogo teste, foi o fato de poder-se simular computação imprecisa com as instruções de tempo real propostas de modo a adaptar a renderização do jogo ao estado atual da máquina. Para tal, fez-se um exemplo em que quando não fosse possível manter-se o nível de renderização, fazia-se uma troca de objetos complexos por caixas. Tal processo pode ser observado na figura 4.12 e demonstra a possibilidade de fazer-se um gerenciador gráfico adaptativo, no qual a capacidade de renderização da máquina alvo é modificada dinamicamente de acordo com suas capacidades e a utilização de processamento atual.

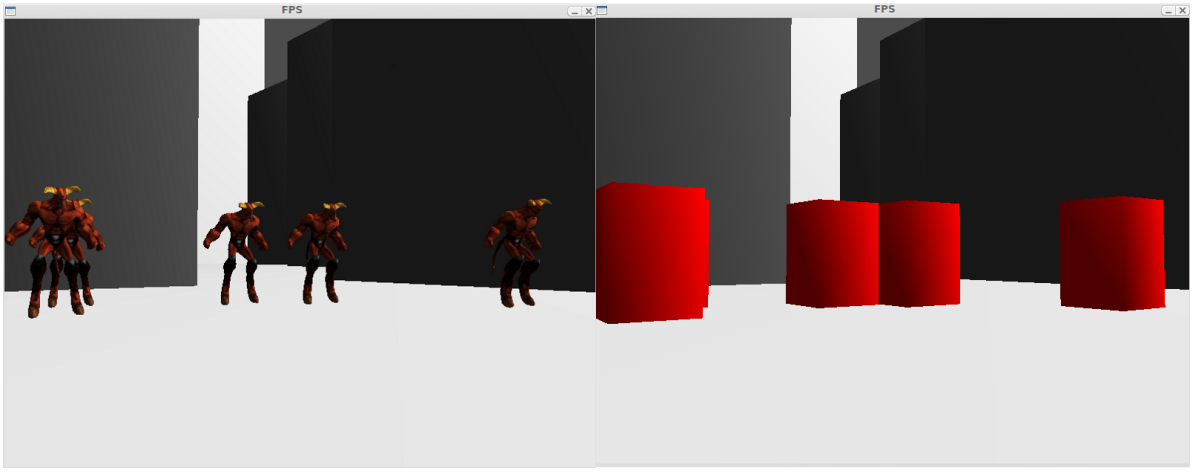

Figura 4.12: Troca de objeto complexo para caixa simples

4.9 IDE para CEx

Para a necessidade cada vez mais crescente de recursos que agilizem e aumentem a produtividade num ambiente de desenvolvimento surgiram as IDEs (*Integrated Desktop Environments*). Elas facilitam o processo de desenvolvimento pela utilização das mais variadas capacidades. Os recursos mais comuns:

- Refatorar código (até mesmo entre projetos distintos)
- Gerar métodos inexistentes
- Autocompletar código
- Análise léxica, sintática e até semântica em tempo real
- Destaque de sintaxe
- Navegação semântica (hierarquia de objetos, etc) pelos projetos
- Integração com ferramentas de controle de versão e de desenvolvimento colaborativo
- Capacidade de gerenciamento de múltiplos projetos

Tendo em vista as grandes vantagens que pode-se obter com a utilização de IDEs para o desenvolvimento, fez-se uma implementação da extensão sobre uma IDE já existente e amplamente usada: o Eclipse(Yang e Jiang 2007).

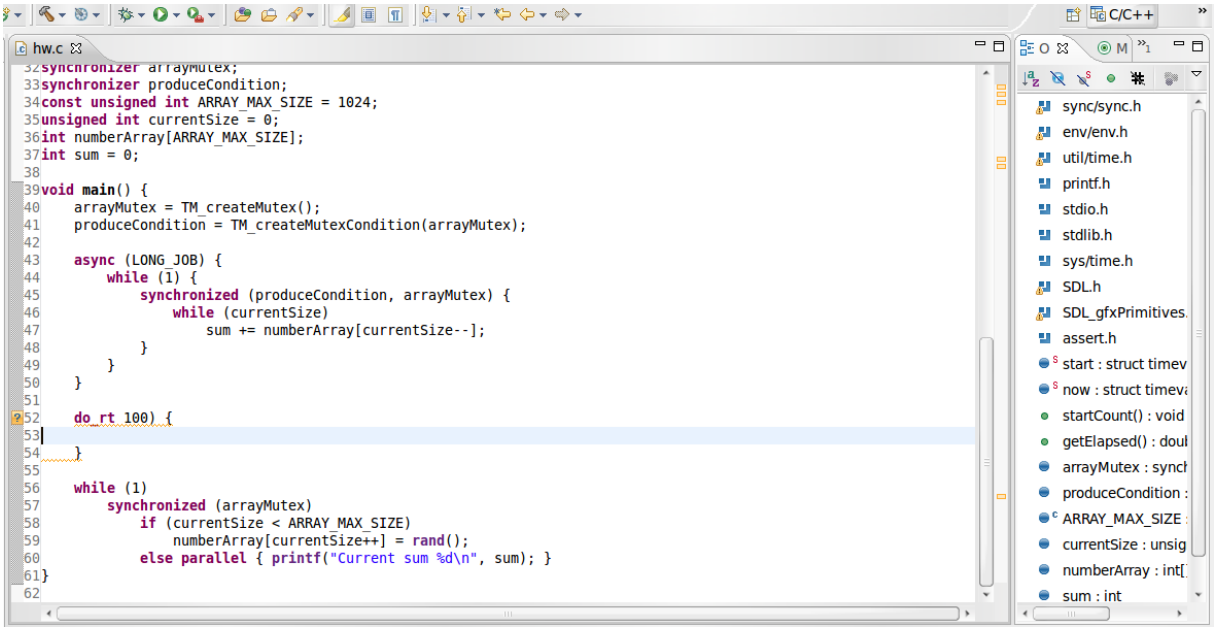

Figura 4.13: Codificando em CEX com o Eclipse utilizando o plugin CDT modificado

4.9.1 Implementação da sintaxe no CDT

O Eclipse é uma IDE opensource cuja maior parte de seu código fonte é escrita na linguagem Java. Essa IDE possui inúmeros plugins que permitem programar nas mais variadas linguagens. Entre esses plugins encontra-se o CDT que é um plugin que permite desenvolvimento em C/C++ com o Eclipse. O plugin também é escrito em Java e possui todas as facilidades citadas na seção anterior e mais algumas não enumaradas. Dado a capacidade interessante e a atualidade desse plugin, decidiu-se implementar a sintaxe da linguagem nele também, facilitando o desenvolvimento de larga escala utlizando CEx.

Para implementar-se a linguagem no CDT, precisou-se espelhar a implementação feita para o Clang, a qual utiliza a *Abstract Syntax Tree* (AST) do LLVM na AST do CDT (própria). Para que também as principais construções da linguagens fossem destacadas na sintaxe, utilizou-se delas como se fossem tipos primitivos, apesar de a especificação não prever. Isso não têm nenhum efeito sintático sobre a linguagem, mas torna mais simples observar quando o tipo de uma variável é algo primitivo da CEx, como a estrutura synchronizer. Na figura 4.13 mostra-se um código escrito em CEx sofrendo destaque de sintaxe e análise sintática pelo plugin modificado. A parte sublinhada em amarelo demonstra que há um erro sintático devido ao esquecimento de um parênteses.

5 *Conclusões e trabalhos futuros*

Esse capítulo visa abordar as principais conclusões obtidas no trabalho, assim como as principais perspectivas de trabalhos futuros possíveis e trabalhos propostos incompletos, observando uma possível implementação para eles.

5.1 Conclusões obtidas

Ao analisar-se as linguagens de tempo real e paralelismo existentes, observou-se a grande necessidade de instruções que facilitassem o tratamento de ambos. Dessa grande necessidade surgiram as palavras-chave e instruções da linguagem: parallel, parallel_for, async, atomic e synchronized. Essas instruções foram pensadas de modo a facilitar situações comuns e para fazer com que a expressividade de paralelização e temporização, presente nas bibliotecas da maioria das linguagens, mantenha-se ao mesmo tempo que haja um ganho interessante de produtividade.

As outras definições da linguagem que são sintaxes alternativas para for (e parallel_for) e as estruturas de dados básicas foram pensadas visando amenizar as deficiências comuns em linguagens simples como C.

A partir da especificação das instruções foi possível criar produções na gramática básica do C[1] e implementá-las com êxito em um compilador, no caso o Clang, dando a essas definições o nome de CEx. Para a implementação das instruções houveram diferentes estratégias, que puderam ser agrupadas em grupos de mesma solução e explanadas. A semântica das instruções também foram definidas por meio de exemplos simples e diretos, de forma mais intuitiva.

Por meio da implementação realizada com o Clang, realizou-se vários testes que demonstram a eficácia geral das instruções rodando em vários ambientes diferentes. Esses testes foram: a sequência de fibonacci que demonstrou a utilização de um laço paralelizado no lugar de um laço comum, o jantar dos filósofos que demonstrou a utilização de *jobs* e de *mutexes*, a ren-

[1] Em notação Yacc (LALR) no capítulo de anexos

derização dos conjuntos de Mandelbrot e os mesmos com renderização dependente de carga, demonstrando o ganho de paralelização na renderização de elementos e a utilização de instruções de tempo real. Também demonstrou-se o exemplo de teste do módulo de física da *engine* Ronin portado para CEx, o qual demonstra a utilização de várias funções da API gráfica e dos mais diversos módulos da *engine*.

Nos testes foi possível observar o ganho de eficiência com a paralelização e também pode ser observado a utilização da extensão para as mais diversas aplicações e não somente para programação de jogos (cujo o teste do módulo de física é o mais relacionado), assim como a facilidade, em relação à bibliotecas padrões da linguagem, de codificar-se para programação concorrente e tempo real no geral.

5.2 Trabalhos incompletos

Dado a extensão do trabalho e o pouco tempo para a entrega do relatório final, decidiu-se que certos elementos não vitais para a linguagem e para o funcionamento dos testes mais interessantes não seriam implementados de modo a implementar com segurança e corretude as instruções desejadas. Os trabalhos incompletos são os módulos não implementados da *engine* e a implementação completa da gramática formal e eles são, no contexto desse projeto, trabalhos futuros.

5.2.1 Finalização dos módulos de som, rede e edição

Os módulos de som, rede e edição foram os principais elementos faltantes para a versão final da *engine* que foi utilizada nesse trabalho.

5.2.2 Módulo de som

O módulo de som deve ser capaz de fazer *stream* de alguns tipos de formatos de aúdio[2] e providenciar o *streaming*, em tempo real, dos referentes. Para implementá-lo poderia-se utilizar de algumas bibliotecas básicas e multi-plataformas de som como SDL_sound aliada a OpenAL. A SDL_sound é capaz de decodificar formatos padrões básicos de aúdio, enquanto o OpenAL faz streaming de aúdio 3D. Unindo ambas é possível fazer uma biblioteca consistente para reprodução de aúdio. O OpenAL possui em sua API a característica de *streaming* de aúdio

[2]Formatos como mp3, ogg, wav...

3D, levando em consideração a fonte do som, os dados a serem sintetizados e a localização do ouvinte. Isso o torna perfeito para programação de jogos, principalmente os 3D.

5.2.3 Módulo de rede

O módulo de rede deve se responsabilizar por manter a sincronização dos objetos equivalentes em diferentes execuções remotas do mesmo jogo. Tal sincronização deve ser otimizada para evitar o envio de dados redundantes, uma vez que o gargalo na maioria das vezes é o tráfego de dados na rede, e caso a quantidade de dados seja grande, a latência das atualizações podem degradar consideravelmente a experiência do jogador. O módulo de rede também deve auxiliar na conexão e desconexão de jogadores, bem como no envio de mensagens internas do jogo.

5.2.4 Módulo de edição

O editor gráfico de mapas e objetos para a engine, poderia ser codificado utilizando-a, mas para tanto, ela necessitaria de um formato de mapa, o qual poderia ser emprestado de alguma outra engine ou próprio. A maioria das *engines* comerciais possuem editores próprios, porém é muito comum que programas de terceiros como o Maya ou 3DMax para editar objetos e mapas. Portanto, seria mais simples ou utilizar de um formato que alguma engine gere, ou ainda criar/reutilizar um plugin para algum programa especializado em modelagem como os citados. Ainda assim, é possível codificar-se um editor inteiro próprio e é a solução mais industrial, uma vez que assim adquiri-se independência em relação aos programas e formatos de terceiros.

5.3 Implementação completa da especificação formal

Alguns elementos foram faltantes na implementação da linguagem, mas como eles eram somente facilidades e não tão necessários como outras partes, decidiu-se não implementá-los.

5.3.1 Sintaxe alternativa para for e parallel_for

As sintaxes alternativas para o parallel_for e for são descritas no capítulo de definições. Elas são mais comportadas no sentido de apresentarem uma condição menos extensa e apresentarem começo e fim bem definidos. Apesar disso, a mesma funcionalidade é provida pelo for e parallel_for comuns, portanto essa sintaxe foi deixada de lado inicialmente. Mesmo assim,

a diferença seria apenas no parsing e semântica. A geração de código seria idêntica, podendo aproveitar as estruturas já existentes para fazê-lo.

5.3.2 Implementação das estruturas de dados propostas na sintaxe

As estruturas de dados da linguagem teriam que ser adicionadas como palavras-chaves privadas da linguagem. No parsing armazenaria-se o tipo de elemento contido. Toda função que retorna algo seria convertida (casting) para o tipo esperado da estrutura de dados e toda chamada para os métodos das estruturas também. Para tal, é necessário que sejam mapeados os métodos existentes das estruturas e toda vez que seja identificado uma chamada fazer as conversões necessárias. Esse método é custoso a nível de implementação. Portanto decidiu-se manter o uso de estruturas de dados no estilo C mesmo.

5.3.3 Palavra-chave atomic

A palavra-chave atomic definida pode ser substituída pela utilização de um mutex associado a variável, por meio da instrução synchronized. Apesar da sintaxe não ser tão limpa, o comportamento dessa forma é mais constante para os operadores. Para implementá-la via Clang, basta adicionar a *keyword* nos tokens reconhecidos pela parte léxica, fazer as verificações semânticas de possíveis tipos suportados e modificar a geração de código dos statementes relacionados à expressões de modo a gerar código de máquina com instruções atômicas (caso a máquina suporte).

5.4 Possíveis trabalhos futuros

Nessa seção vislumbra-se os possíveis rumos a serem tomados a partir da base constituída por esse trabalho e como poderia começar-se cada uma das idéias referentes. Primeiramente, vislumbra-se implementar os elementos incompletos e faltantes citados nas últimas seções. Contudo, além desses citados, ainda existem mais alguns outros, não contemplados inicialmente pelo escopo do trabalho, mas que são passíveis de implementação. São eles: complemento do renderizador multithread, módulos de IA previstos na interface da *engine* e da interface RN (CEx), melhoria de mensagens de erros semânticos nas instruções complexas da linguagem, suporte a acesso de variáveis na pilha para o parallel_for, suporte para sintaxes alternativas imprevistas, implementação como uma extensão de C++ e modo de execução com depuração avançada via biblioteca intermediária.

Número de objetos / Renderizador	10.000	100.000
Sequencial	742,7	55,2
Multithread	611,4	58,5

Tabela 5.1: Comparação de performance, em quadros por segundo, do renderizador sequencial e multithread

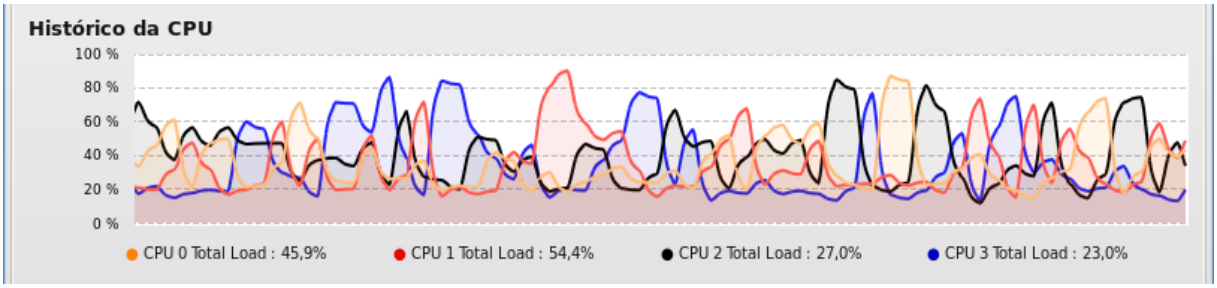

Figura 5.1: Carga de processamento durante a renderização em múltiplas threads

5.4.1 Complemento do renderizador multithread

Da forma que foi implementado, o módulo de renderização multithread não obteve o ganho de performance esperado. A implementação atual dificilmente consegue manter as threads the viewclipping ocupadas o suficiente para obter-se ganho significativo de performance e frequentemente até mesmo apresentando uma performance inferior a do renderizador sequencial.

O benchmark da tabela 5.1 compara a performance atual do renderizador sequencial e do multithread ao renderizar uma cena contendo número variado de objetos. Os testes foram feitos em um computador com uma CPU Intel(R) Core(TM)2 Quad Q8200 2.33GHz e uma placa de vídeo GeForce 9600GT. Os valores dos estes estão em frames por segundo (média de dez segundos).

Na figura 5.1 onde é demostrada a carga dos 4 processadores durante a re-execução do teste com 100.000 objetos no renderizador multithread fica claro o motivo da pobre performance do renderizador multithread.

5.4.2 Módulo de inteligência artificial

Assim como o módulo de física, o módulo de inteligência artificial proveria uma interface para acesso algoritmos de inteligencia artificial implementação de uma IA nos jogos. A engine proveria algoritmos de path finding, steering behaviors (desvio de objetos móveis) entre outros.

5.4.3 Efeitos gráficos pós-processados

Os efeitos gráficos pós-processados, adicionam elementos gráficos simulados por meio de programação de shaders e técnicas visuais. Entre eles estão a simulação de fogo e fumaça, água, refração da luz, mapemanento de sombras, luzes dinâmicas entre outros. Vários desses efeitos encontram-se implementados genericamente em motores proprietários e/ou comerciais.

5.4.4 Melhoria de erros semânticos nas instruções

As instruções parallel, parallel_for e async são definidas dentro do escopo de uma função, contudo elas não podem, na definição dada, acessar as variáveis de pilha. Quando o programador o faz, um erro semântico não é gerado, pois as variáveis, na utilização encontram-se no escopo da função que usa a instrução. No código gerado, o corpo das instruções gera uma outra função e os acessos à variáveis que eram antes locais, agora irão gerar erros de backend. Esses erros podem ser melhorados ao verificar-se que uma determinada variável declarada na função está sendo acessada na instrução alvo, gerando assim um erro semântico e não um erro de backend. Uma implementação desse tipo, apesar de não trivial, auxiliaria o programador a entender problemas recorrentes em grandes códigos utilizando as instruções.

5.4.5 Suporte para acesso à stack no parallel_for

Conforme observado no capítulo de prototipagem, o parallel_for não deve acessar a stack local da função que o utiliza. Isso deve-se ao fato de ele obedecer a uma das regras básicas de códigos concorrentes: uma thread tem acesso somente a sua pilha. Contudo, em determinadas ocasiões, é interessante como comodidade que o programador seja capaz de utilizar alguma variável na função chamada. Nesse caso, não haveria problema algum em acessar as variáveis da função quando o comportamento atual do parallel_for é esperar o término dos laços para cada utilização. Porém, isso é problemático, dado que o programador pode mudar o "comportamento de espera" para o laço paralelizado.

Uma outra solução seria copiar as variáveis da stack passando-as com o mesmo nome como argumentos da função forjada via *vargs* (cuja sintaxe em C é "..."). Assim, teria-se um funcionamento mais interessante, independente da função chamadora, porém com um problema grave: variáveis do tipo array declaradas na pilha seriam inteiramente copiadas ao passá-las como parâmetro.

Mesmo que a segunda solução parece ser melhor, por ter menor dependência com a fun-

ção da instrução, nenhuma das duas soluções funciona perfeitamente talvez fosse interessante estudar-se uma terceira, caso exista.

5.4.6 Suporte à sintaxes alternativas

Em algumas instruções da linguagem como a async e a synchronized, a necessidade de utilizar-se o nome do identificador como elemento faz com que não seja possível utilizar expressões. As expressões são interessantes porque permitem acesso direto à arranjos e utilizar um retorno de uma função como argumento para a instrução. Esse tipo de melhoria não foi vislumbrado inicialmente, mas com o uso das instruções propostos essas possíveis melhorias foram observadas.

O tratamento semântico seria basicamente o mesmo, porém seria somente necessário também averiguar se o resultado das expressões de argumento conferem com o tipo esperado para a instrução.

5.4.7 Implementar a extensão C++Ex

C é uma linguagem voltada para programação de sistemas. C++ é utilizada no meio de jogos e por possuir uma sintaxe mais "relaxada" e existirem muitas bibliotecas e funcionalidades para ela e por ser uma linguagem orientada à objetos. Ela possui muitas capacidades faltantes em C como sobrescrita de operadores para objetos, possibilidade de meta-programação e estruturas de dados básicas presentes em sua biblioteca padrão.

Um outro fator interessante é que todos os códigos da interface CEx são implementados em C++, assim como a engine Ronin. Uma extensão implementada diretamente nela, possibilitaria remover-se grande parte da latência devido as chamadas indiretas pela interface, como também diminuiria a quantidade de código a ser mantido. Também, as estruturas de dados propostas poderiam ser facilmente implementadas utilizando-se de meta-programação via templates e algumas propostas já estão até mesmo na biblioteca padrão de C++.

5.4.8 Modo de execução com depuração avançada

A idéia de haver um modo especial de depuração é que seria possível no ThreadMixer obter-se informações da execução atual, possibilitando detectar deadlocks e tratá-los quando possível, assim como observar de maneira mais abstrata o funcionamento de um programa qualquer. Isso porque seria possível fazer algum tipo de ferramenta que fosse depurando, até de forma visual,

o código e seus múltiplos fluxos de execução. Tal ferramenta seria útil por inúmeras razões e poderia ser utilizada em conjunto com as já existentes para C como o gdb, de modo a tornar mais simples a solução de problemas e observação de gargalos de execução.

Referências Bibliográficas

Akenine-Möller, Haines e Hoffman 2008 AKENINE-MöLLER, Tomas; HAINES, Eric; HOFFMAN, Natty. *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., 2008. 1045 p. ISBN 987-1-56881-424-7.

Alves 2008 ALVES, Felipe Borges. *Engine Ronin*. Novembro 2008. Disponível em: <https://svn.inf.ufsc.br/dedefbs/Ronin/trunk>.

Armstrong 2003 ARMSTRONG, Joe. *Making Reliable Distributed Systems in the Presence of Software Errors*. 2003. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.3.408>.

Bacon, Cheng e Rajan 2003 BACON, David F.; CHENG, Perry; RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 38, n. 1, p. 285–298, 2003. ISSN 0362-1340.

Bethke 2003 BETHKE, Erik. *Game Development and Production*. [S.l.]: Worlware Publishing, 2003. ISBN 1556229518.

Board et al. 2007 BOARD, OpenGL Architecture Review; SHREINER, Dave; WOO, Mason; NEIDER, Jackie; DAVIS, Tom. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2.1*. [S.l.]: Addison-Wesley Professional, 2007. ISBN 0321481003, 9780321481009.

Boehm 2008 BOEHM, Mirko. *ThreadWeaver: ThreadWeaver*. KDE, Novembro 2008. Disponível em: <http://www.englishbreakfastnetwork.org/apidocs/apidox-kde-4.0/kdelibs-apidocs/threadweaver/html/index.html>.

Boeing e Bräunl 2007 BOEING, Adrian; BRäUNL, Thomas. Evaluation of real-time physics simulation systems. In: *GRAPHITE '07: Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*. New York, NY, USA: ACM, 2007. p. 281–288. ISBN 978-1-59593-912-8.

Buttazzo 2004 BUTTAZZO, Giorgio C. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Santa Clara, CA, USA: Springer-Verlag TELOS, 2004. ISBN 0387231374.

Chen e Cheng 2005 CHEN, Bo; CHENG, Harry H. Interpretive opengl for computer graphics. *Computers & Graphics*, v. 29, n. 3, p. 331–339, June 2005. Disponível em: <http://dx.doi.org/10.1016/j.cag.2005.03.002>.

Cilk++ 2009 CILK++. Junho 2009. Disponível em: <http://www.cilk.com>.

Clang 2009 CLANG. *"clang" C Family Frontend for LLVM*. Maio 2009. Disponível em: <http://clang.llvm.org/features.html>.

Dagum e Menon 1998 DAGUM, L.; MENON, R. Openmp: an industry standard api for shared-memory programming. *Computational Science and Engineering, IEEE*, v. 5, n. 1, p. 46–55, Jan-Mar 1998. ISSN 1070-9924.

Dijkstra 1965 DIJKSTRA, E. W. Solution of a problem in concurrent programming control. *Commun. ACM*, ACM, New York, NY, USA, v. 8, n. 9, p. 569, 1965. ISSN 0001-0782.

Eppstein, Goodrich e Sun 2005 EPPSTEIN, David; GOODRICH, Michael T.; SUN, Jonathan Z. The skip quadtree: a simple dynamic data structure for multidimensional data. In: *SCG '05: Proceedings of the twenty-first annual symposium on Computational geometry*. New York, NY, USA: ACM, 2005. p. 296–305. ISBN 1-58113-991-8.

Farias et al. 2005 FARIAS, Thiago Souto Maior Cordeiro de; PESSOA, Saulo Andrade; TEICHRIEB, Veronica; KELNER, Judith. O engine gráfico ogre. *SBGames 2005 parto de IV Workshop Brasileiro de Jogos e Entretenimento Digital*, São Paulo, SP, Brasil, p. 23–25, Novembro 2005.

Flynn 1972 FLYNN, M. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C-21, p. 948+, 1972. Disponível em: <http://en.wikipedia.org/wiki/Flynn's taxonomy>.

Gamma et al. 1994 GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John M. *Design Patterns: Elements of Reusable Object-Oriented Software*. illustrated edition. Addison-Wesley Professional, 1994. Hardcover. ISBN 0201633612. Disponível em: <http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612>.

Gehani 1991 GEHANI, N. H. Concurrent c: real-time programming and fault tolerance. *Softw. Eng. J.*, Michael Faraday House, Herts, UK, UK, v. 6, n. 3, p. 83–92, 1991. ISSN 0268-6961.

Gehani e Roome 1988 GEHANI, N. H.; ROOME, W. D. Concurrent c++: concurrent programming with class(es). *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 18, n. 12, p. 1157–1177, 1988. ISSN 0038-0644.

Gesser 2002 GESSER, Carlos Eduardo. Ambiente para geração de analisadores léxicos e sintáticos. Novembro 2002.

Goldberg e Robson 1983 GOLDBERG, Adele; ROBSON, David. *Smalltalk-80: the language and its implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN 0-201-11371-6.

Hoare 1974 HOARE, C. A. R. Monitors: an operating system structuring concept. *Commun. ACM*, ACM, New York, NY, USA, v. 17, n. 10, p. 549–557, 1974. ISSN 0001-0782.

IBM 1998 IBM. *Pthreads APIs - User's Guide and Reference*. IBM Corporation, 1998. Disponível em: <http://cs.pub.ro/ apc/2003/resources/pthreads/uguide/document.htm>.

idSoftware 2005 IDSOFTWARE. *Quake3 Raytraced*. Julho 2005. Disponível em: <http://www.idfun.de/q3rt/>.

IEEE 2004 IEEE. *Standard for information technology - portable operating system interface (POSIX). Shell and utilities*. [S.l.], 2004. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1309816>.

ISO 1999 ISO. *ISO C Standard 1999*. [S.l.], 1999. ISO/IEC 9899:1999 draft. Disponível em: <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf>.

Kessenich, Baldwin e Rost 2008 KESSENICH, John; BALDWIN, Dave; ROST, Randi. *The OpenGL Shading Language*. Agosto 2008. Disponível em: <www.opengl.org/registry/doc/GLSLangSpec.Full.1.30.08.pdf>.

Lattner e Adve 2004 LATTNER, Chris; ADVE, Vikram. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California: [s.n.], 2004.

Lesk e Schmidt 1975 LESK, M. E.; SCHMIDT, E. *Lex - A Lexical Analyzer Generator*. [S.l.], 1975.

Liu e Layland 1973 LIU, C. L.; LAYLAND, James W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, ACM, New York, NY, USA, v. 20, n. 1, p. 46–61, 1973. ISSN 0004-5411.

Luebke et al. 2004 LUEBKE, David; HARRIS, Mark; KRüGER, Jens; PURCELL, Tim; GOVINDARAJU, Naga; BUCK, Ian; WOOLLEY, Cliff; LEFOHN, Aaron. Gpgpu: general purpose computation on graphics hardware. In: *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*. New York, NY, USA: ACM Press, 2004. Disponível em: <http://dx.doi.org/10.1145/1103900.1103933>.

Luna 2003 LUNA, Frank D. *Introduction to 3d Game Programming with Directx 9.0*. Plano, TX, USA: Wordware Publishing Inc., 2003. ISBN 1556229135.

Mark et al. 2003 MARK, William R.; GLANVILLE, R. Steven; AKELEY, Kurt; KILGARD, Mark J. Cg: a system for programming graphics hardware in a c-like language. In: *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*. New York, NY, USA: ACM, 2003. p. 896–907. ISBN 1-58113-709-5.

Matheson 2008 MATHESON, Lesley. *Gamasutra - High Impact's Matheson: Build, Don't Buy Game Engines*. Maio 2008. Disponível em: <http://www.gamasutra.com/php-bin/news index.php?story=18686>.

Parr e Quong 1995 PARR, Terence J.; QUONG, Russell W. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, v. 25, p. 789–810, 1995.

Peter, Buhr e Sartipi 1996 PETER, Prof; BUHR, A.; SARTIPI, Kamran. *Concurrent C/C++ Programming Languages*. Abril 1996.

Phaethon 2003 PHAETHON. *Quake 3 Virtual Machine (Q3VM) Specifications*. Fevereiro 2003. Disponível em: <http://icculus.org/ phaethon/q3mc/q3vm specs.html>.

Physics Simulation Forum 2009 PHYSICS Simulation Forum. 2009. Disponível em: <http://www.bulletphysics.com>.

Quake-C 2008 QUAKE-C. *Quake-C Specifications v1.0*. Novembro 2008. Disponível em: <http://www.inside3d.com/qcspecs/qc-menu.htm>.

Real Time Specification for Java REAL Time Specification for Java. Disponível em: <http://www.rtsj.org>.

Rost 2006 ROST, Randi J. *OpenGL Shading Language*. [S.l.]: Addison Wesley, 2006. 800 p. ISBN 978-0-321-33489-3.

Segal e Akeley 2008 SEGAL, Mark; AKELEY, Kurt. *The OpenGL Graphics System: A Specification*. Agosto 2008. Disponível em: <www.opengl.org/registry/doc/glspec30.20080811.pdf>.

Shishikura 1991 SHISHIKURA, Mitsuhiro. The hausdorff dimension of the boundary of the mandelbrot set and julia sets. Apr 1991. Disponível em: <http://arxiv.org/abs/math.DS/9201282>.

Tanenbaum, Sharp e A 1992 TANENBAUM, Andrew S.; SHARP, Gregory J.; A, De Boelelaan. *Modern operating systems*. [S.l.]: Prentice Hall, 1992. ISBN 9780135881873.

Yang e Jiang 2007 YANG, Zhihui; JIANG, Michael. Using eclipse as a tool-integration platform for software development. *IEEE Softw.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 24, n. 2, p. 87–89, March 2007. ISSN 0740-7459. Disponível em: <http://dx.doi.org/10.1109/MS.2007.58>.

6 Anexos

6.1 Produções e terminais adicionados (notação YACC)

```
compound_statement
:
 | FOR '(' TYPE_NAME IDENTIFIER ':' IDENTIFIER ')' statement
 | PARALLEL_FOR '(' TYPE_NAME IDENTIFIER ':' '[' CONSTANT ',' CONSTANT ']' ')' statement
 | PARALLEL_FOR '(' TYPE_NAME IDENTIFIER ':' IDENTIFIER ')' statement
 | PARALLEL_FOR '(' expression_statement expression_statement ')' statement
 | PARALLEL_FOR '(' expression_statement expression_statement expression ')' statement
 ;

data_struct_type
 : '<' data_struct_type_specifier '>'
 ;

data_struct_type_specifier
 : type_specifier
 | pointer type_specifier
 | CONST pointer type_specifier
 ;

data_struct
 : LIST data_struct_type
 | VECTOR data_struct_type
 | HASH_MAP data_struct_type
 | TREE_MAP data_struct_type
 | HASH_SET data_struct_type
 | TREE_SET data_struct_type
 ;

function_definition
 : function_definition_sync
 | function_definition_async
 ;

function_definition_sync
: declaration_specifiers declarator declaration_list compound_statement
| declaration_specifiers declarator compound_statement
| declarator declaration_list compound_statement
| declarator compound_statement
;
```

```
function_definition_async
 : PARALLEL function_definition_sync
 | ASYNC '(' IDENTIFIER ')' function_definition_sync
 ;


declarator
:
 | direct_declarator '[' constant_expression  ':' constant_expression ']'
 | synchronized_statement
 | parallel_statement
 | rt_statement
;

parallel_statement
 : PARALLEL statement
 | ASYNC '(' IDENTIFIER ')' statement
;

primary_expression
:
 | postfix_expression '[' expression ':' expression ']'
 | data_struct
;

rt_statement
 : DO_RT '(' constant_expression ')' statement
 ;

synchronized_statement
 : SYNCHRONIZED '(' identifier_list ')' statement
 ;

type_qualifier
:
 | ATOMIC
;
```

6.2 Gramática completa na notação YACC

```
%TOKEN ASYNC PARALLEL PARALLEL_FOR SYNCHRONIZED ATOMIC TASK
%TOKEN LIST VECTOR HASH_MAP TREE_MAP HASH_SET TREE_SET

%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token STRUCT UNION ENUM ELLIPSIS
```

```
%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN DO_RT

%start translation_unit
%%

abstract_declarator
    : pointer
    | direct_abstract_declarator
    | pointer direct_abstract_declarator;

additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression;

and_expression
    : equality_expression
    | and_expression '&' equality_expression;

argument_expression_list
    : assignment_expression
    | argument_expression_list ',' assignment_expression;

assignment_expression
    : conditional_expression
    | unary_expression assignment_operator assignment_expression;

assignment_operator
    : '='
    | MUL_ASSIGN
    | DIV_ASSIGN
    | MOD_ASSIGN
    | ADD_ASSIGN
    | SUB_ASSIGN
    | LEFT_ASSIGN
    | RIGHT_ASSIGN
    | AND_ASSIGN
    | XOR_ASSIGN
    | OR_ASSIGN;

cast_expression
    : unary_expression
    | '(' type_name ')' cast_expression;

conditional_expression
    : logical_or_expression
    | logical_or_expression '?' expression ':' conditional_expression;

constant_expression
    : conditional_expression;

compound_statement
    : '{' '}'
```

```
    | '{' statement_list '}'
    | '{' declaration_list '}'
    | '{' declaration_list statement_list '}';

data_struct_type
    : '<' data_struct_type_specifier '>';

data_struct_type_specifier
    : type_specifier
    | pointer type_specifier
    | CONST pointer type_specifier;

data_struct
    : LIST data_struct_type
    | VECTOR data_struct_type
    | HASH_MAP data_struct_type
    | TREE_MAP data_struct_type
    | TREE_SET data_struct_type
    | HASH_SET data_struct_type;

declaration
    : declaration_specifiers ';'
    | declaration_specifiers init_declarator_list ';';

declaration_list
    : declaration
    | declaration_list declaration;

declaration_specifiers
    : storage_class_specifier
    | storage_class_specifier declaration_specifiers
    | type_specifier
    | type_specifier declaration_specifiers
    | type_qualifier
    | type_qualifier declaration_specifiers;

declarator
    : pointer direct_declarator
    | direct_declarator;

direct_abstract_declarator
    : '(' abstract_declarator ')'
    | '[' ']'
    | '[' constant_expression ']'
    | direct_abstract_declarator '[' ']'
    | direct_abstract_declarator '[' constant_expression ']'
    | '(' ')'
    | '(' parameter_type_list ')'
    | direct_abstract_declarator '(' ')'
    | direct_abstract_declarator '(' parameter_type_list ')';

direct_declarator
    : IDENTIFIER
    | '(' declarator ')'
```

```
    | direct_declarator '[' constant_expression ']'
    | direct_declarator '[' constant_expression  ':' constant_expression ']'
    | direct_declarator '[' ']'
    | direct_declarator '(' parameter_type_list ')'
    | direct_declarator '(' identifier_list ')'
    | direct_declarator '(' ')';

enum_specifier
    : ENUM '{' enumerator_list '}'
    | ENUM IDENTIFIER '{' enumerator_list '}'
    | ENUM IDENTIFIER;

enumerator
    : IDENTIFIER
    | IDENTIFIER '=' constant_expression;

enumerator_list
    : enumerator
    | enumerator_list ',' enumerator;

equality_expression
    : relational_expression
    | equality_expression EQ_OP relational_expression
    | equality_expression NE_OP relational_expression;

expression_statement
    : ';'
    | expression ';';

exclusive_or_expression
    : and_expression
    | exclusive_or_expression '^' and_expression;

expression
    : assignment_expression
    | expression ',' assignment_expression;

external_declaration
    : function_definition
    | declaration;

function_definition
    : function_definition_sync
    | function_definition_async;

function_definition_sync
    : declaration_specifiers declarator declaration_list compound_statement
    | declaration_specifiers declarator compound_statement
    | declarator declaration_list compound_statement
    | declarator compound_statement;

function_definition_async
    : PARALLEL function_definition_sync
    | ASYNC '(' IDENTIFIER ')' function_definition_sync;
```

```
inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression '|' exclusive_or_expression;

init_declarator
    : declarator
    | declarator '=' initializer;

init_declarator_list
    : init_declarator
    | init_declarator_list ',' init_declarator;

initializer
    : assignment_expression
    | '{' initializer_list '}'
    | '{' initializer_list ',' '}';

initializer_list
    : initializer
    | initializer_list ',' initializer;

identifier_list
    : IDENTIFIER
    | identifier_list ',' IDENTIFIER;

iteration_statement
    : WHILE '(' expression ')' statement
    | DO statement WHILE '(' expression ')' ';'
    | FOR '(' expression_statement expression_statement ')' statement
    | FOR '(' expression_statement expression_statement expression ')' statement
    | FOR '(' TYPE_NAME IDENTIFIER ':' IDENTIFIER ')' statement
    | PARALLEL_FOR '(' expression_statement expression_statement ')' statement
    | PARALLEL_FOR '(' expression_statement expression_statement expression ')' statement
    | PARALLEL_FOR '(' TYPE_NAME IDENTIFIER ':' '[' CONSTANT ',' CONSTANT ']' ')' statement
    | PARALLEL_FOR '(' TYPE_NAME IDENTIFIER ':' IDENTIFIER ')' statement;

jump_statement
    : GOTO IDENTIFIER ';'
    | CONTINUE ';'
    | BREAK ';'
    | RETURN ';'
    | RETURN expression ';';

labeled_statement
    : IDENTIFIER ':' statement
    | CASE constant_expression ':' statement
    | DEFAULT ':' statement;

logical_and_expression
    : inclusive_or_expression
    | logical_and_expression AND_OP inclusive_or_expression;

logical_or_expression
```

```
    : logical_and_expression
    | logical_or_expression OR_OP logical_and_expression;

multiplicative_expression
    : cast_expression
    | multiplicative_expression '*' cast_expression
    | multiplicative_expression '/' cast_expression
    | multiplicative_expression '%' cast_expression;

parallel_statement
    : PARALLEL statement
    | ASYNC '(' IDENTIFIER ')' statement;

parameter_type_list
    : parameter_list
    | parameter_list ',' ELLIPSIS;

parameter_list
    : parameter_declaration
    | parameter_list ',' parameter_declaration;

parameter_declaration
    : declaration_specifiers declarator
    | declaration_specifiers abstract_declarator
    | declaration_specifiers;

pointer
    : '*'
    | '*' type_qualifier_list
    | '*' pointer
    | '*' type_qualifier_list pointer;

postfix_expression
    : primary_expression
    | postfix_expression '[' expression ']'
    | postfix_expression '[' expression ':' expression ']'
    | postfix_expression '(' ')'
    | postfix_expression '(' argument_expression_list ')'
    | postfix_expression '.' IDENTIFIER
    | postfix_expression PTR_OP IDENTIFIER
    | postfix_expression INC_OP
    | postfix_expression DEC_OP;

primary_expression
    : IDENTIFIER
    | CONSTANT
    | STRING_LITERAL
    | '(' expression ')';

relational_expression
    : shift_expression
    | relational_expression '<' shift_expression
    | relational_expression '>' shift_expression
    | relational_expression LE_OP shift_expression
```

```
    | relational_expression GE_OP shift_expression;

rt_statement
    : DO_RT '(' constant_expression ')' statement
    | PERIODIC_RT '(' constant_expression ')' '(' constant_expression ')' statement

selection_statement
    : IF '(' expression ')' statement
    | IF '(' expression ')' statement ELSE statement
    | SWITCH '(' expression ')' statement;

shift_expression
    : additive_expression
    | shift_expression LEFT_OP additive_expression
    | shift_expression RIGHT_OP additive_expression;

specifier_qualifier_list
    : type_specifier specifier_qualifier_list
    | type_specifier
    | type_qualifier specifier_qualifier_list
    | type_qualifier;

statement
    : labeled_statement
    | compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    | synchronized_statement
    | parallel_statement
    | rt_statement;

statement_list
    : statement
    | statement_list statement;

storage_class_specifier
    : TYPEDEF
    | EXTERN
    | STATIC
    | AUTO
    | REGISTER;

struct_declarator_list
    : struct_declarator
    | struct_declarator_list ',' struct_declarator;

struct_declarator
    : declarator
    | ':' constant_expression
    | declarator ':' constant_expression;
```

```
struct_or_union_specifier
    : struct_or_union IDENTIFIER '{' struct_declaration_list '}'
    | struct_or_union '{' struct_declaration_list '}'
    | struct_or_union IDENTIFIER;

struct_or_union
    : STRUCT
    | UNION;

struct_declaration_list
    : struct_declaration
    | struct_declaration_list struct_declaration;

struct_declaration
    : specifier_qualifier_list struct_declarator_list ';';

synchronized_statement
    : SYNCHRONIZED '(' identifier_list ')' statement;

translation_unit
    : external_declaration
    | translation_unit external_declaration;

type_name
    : specifier_qualifier_list
    | specifier_qualifier_list abstract_declarator;

type_qualifier_list
    : type_qualifier
    | type_qualifier_list type_qualifier;

type_qualifier
    : CONST
    | VOLATILE
    | ATOMIC;

type_specifier
    : VOID
    | CHAR
    | SHORT
    | INT
    | LONG
    | FLOAT
    | DOUBLE
    | SIGNED
    | UNSIGNED
    | struct_or_union_specifier
    | enum_specifier
    | data_struct
    | TYPE_NAME;

unary_expression
    : postfix_expression
    | INC_OP unary_expression
```

```
    | DEC_OP unary_expression
    | unary_operator cast_expression
    | SIZEOF unary_expression
    | SIZEOF '(' type_name ')';

unary_operator
    : '&'
    | '*'
    | '+'
    | '-'
    | '~'
    | '!';
```

6.3 Biblioteca básica

Nessa seção encontram-se as descrições de funções, estruturas e constantes definidas.

6.3.1 Constantes

```
const int SIMPLE_JOB = -1;
const int LONG_JOB = 0;
```

6.3.2 Estruturas

```
typedef struct {
    const char* name;
    unsigned int width, height, bpp;
    int full_screen, show_cursor, grab_input, double_buffer;
} display_cfg;

typedef enum {
    DYNAMIC,
    STATIC
} ObjPhysicalProperty;

typedef struct {
    float r, g, b, a;
} rn_color;

typedef enum {
    LOW,
    MEDIUM,
    HIGH,
    HIGHEST
} rn_texture_quality;

typedef struct {
    float x, y, z;
} rn_point3;

typedef enum {
```

```
    UNCONTINUOUS_COLLISION,
    CONTINUOUS_COLLISION
} rn_simul_tp;


typedef struct {
    float x, y, z;
} rn_vector3;

typedef struct {
    unsigned int width, height;
} size2;

typedef struct {
    unsigned int width, height, depth;
} size3;

typedef enum {
    MUTEX,
    BARRIER,
    SEMAPHORE,
    CONDITION
} sync_tp;

struct synchronizer {
    void* data;
    sync_tp tp;
};

#define VOIDPTR_RN_OBJ_WRAPPER(A) typedef struct { \
    void* data; \
    } rn_##A

#define VOIDPTR_OBJ_WRAPPER(A) struct A { \
    void* data; \
    };\
    typedef struct A A;

#define SMARTORVOIDPTR_RN_OBJ_WRAPPER(A) typedef struct { \
    void* data; \
    RnPtrType tp;\
    } rn_smartptr_##A; \
    VOIDPTR_RN_OBJ_WRAPPER(A)

VOIDPTR_OBJ_WRAPPER(job);

VOIDPTR_RN_OBJ_WRAPPER(camera);
VOIDPTR_RN_OBJ_WRAPPER(clock);
VOIDPTR_RN_OBJ_WRAPPER(collision_detection);
VOIDPTR_RN_OBJ_WRAPPER(display);
VOIDPTR_RN_OBJ_WRAPPER(input);
VOIDPTR_RN_OBJ_WRAPPER(light);
VOIDPTR_RN_OBJ_WRAPPER(object);
VOIDPTR_RN_OBJ_WRAPPER(mesh);
```

```
VOIDPTR_RN_OBJ_WRAPPER(physic_handler);
VOIDPTR_RN_OBJ_WRAPPER(simulation);
VOIDPTR_RN_OBJ_WRAPPER(texture_map);


SMARTORVOIDPTR_RN_OBJ_WRAPPER(box);
SMARTORVOIDPTR_RN_OBJ_WRAPPER(volume);
```

6.3.3 Sincronismo

```
void TM_resizeSimpleJobWorkers(unsigned int);

job TM_createJob();
void TM_execAndWaitJob(job j);

synchronizer TM_createMutex();
void TM_destroyMutex(synchronizer);

void TM_mutexLock(synchronizer mutex);
void TM_mutexUnlock(synchronizer mutex);

synchronizer TM_createBarrier(unsigned int size);
void TM_waitBarrier(synchronizer);
void TM_postSemaphore(synchronizer);
void TM_destroyBarrier(synchronizer);

synchronizer TM_createSemaphore(int initial);
void TM_waitSemaphore(synchronizer);
void TM_destroySemaphore(synchronizer);

synchronizer TM_createMutexCondition(synchronizer mutex);
synchronizer TM_createCondition();
void TM_destroyCondition(synchronizer self);
void TM_waitMutexCondition(synchronizer self, synchronizer mutex);
void TM_waitCondition(synchronizer self);
void TM_notifyOneCondition(synchronizer self);
void TM_notifyAllCondition(synchronizer self);

synchronizer TM_createMutexCondition(synchronizer mutex);
synchronizer TM_createCondition();
void TM_waitMutexCondition(synchronizer, synchronizer mutex);
void TM_waitCondition(synchronizer);
void TM_notifyOneCondition(synchronizer);
void TM_notifyAllCondition(synchronizer);
void TM_destroyCondition(synchronizer);
```

6.3.4 Builtins

```
void __builtin_parallel_for_call(void (*func)(int), job j, int i);

void __builtin_parallel_call(void (*func)(void));

void __builtin_async_named_call(void (*func)(void), const char *name);
```

```
void __builtin_async_id_call(void (*func)(void), int id);

void __builtin_presync(synchronizer sync);
void __builtin_possync(synchronizer mutex);

int __builtin_do_rt_start(unsigned int millis);
void __builtin_do_rt_end(int id);

void __builtin_periodic_rt_call(
        void (*func)(void), unsigned int deadline, unsigned int periodicity);
```

6.3.5 RN - Ronin C Interface

```
/*Box*/
rn_volume RN_box_getAsVolume(rn_box box);
rn_box RN_box_create(rn_vec3 sizeVec);
rn_vec3 RN_box_getSize(rn_box self);

/*Camera*/
rn_point3 RN_camera_getEye(rn_camera self);
rn_vector3 RN_camera_getVpn(rn_camera self);
rn_vector3 RN_camera_getVup(rn_camera self);

void RN_camera_setEye(rn_camera self, rn_point3 e);
void RN_camera_setVpn(rn_camera self, rn_vector3 v);
void RN_camera_setVup(rn_camera self, rn_vector3 v);

void RN_camera_move(rn_camera self, rn_vector3 direction);
void RN_camera_moveHorizontal(rn_camera self, float step);
void RN_camera_moveVertical(rn_camera self, float step);
void RN_camera_moveForward(rn_camera self, float step);

void RN_camera_rotateDegrees(rn_camera self, float angle, rn_vector3 axis);
void RN_camera_rotateRadians(rn_camera self, float angle, rn_vector3 axis);

void RN_camera_verticalRotationDegrees(rn_camera self, float angle);
void RN_camera_verticalRotationRadians(rn_camera self, float angle);

/*Clock*/
rn_clock RN_clock_new(void);
void RN_clock_destroy(rn_clock self);

// static method
uint64_t RN_clock_currentTime(void);

unsigned int RN_clock_getDiffTime(rn_clock self);

float RN_clock_getDt(rn_clock self);

/*CollisionDetection*/
rn_collision_detection RN_collision_detection_create(size3 s);
void RN_collision_detection_destroy(rn_collision_detection self);
```

```
void RN_collision_detection_addCollisionHandler(rn_collision_detection self,
    rn_collision_handler h);
bool RN_collision_detection_removeCollisionHandler(rn_collision_detection self,
    rn_collision_handler h);

rn_collision_handler RN_collision_detection_checkCollision_segment(rn_collision_detection
    self,
        rn_point3 begin, rn_point3 end);

/*Input*/
rn_input RN_input_new(void);
void RN_input_destroy(rn_input self);

void RN_input_pollEvents(rn_input self);

int RN_input_getNumKeys(rn_input self);

int RN_input_getMouseDx(rn_input self);
int RN_input_getMouseDy(rn_input self);

// bt is the button ID from SDL
bool RN_input_isMouseDown(rn_input self, int bt);
bool RN_input_isKeyDown(rn_input self, int key);


/*Light*/
rn_light RN_light_createPoint(rn_point3 pos, rn_color amb, rn_color dif,
        rn_color spec);

rn_light RN_light_createDir(rn_vector3 dir, rn_color amb, rn_color dif,
        rn_color spec);

void RN_light_destroy(rn_light self);

/*Object*/
rn_object RN_object_createFromRenderer(rn_renderer r);
rn_object RN_object_createFromPhysicHandler(rn_physic_handler ph);
rn_object RN_object_createFromBox(rn_box b);

void RN_object_destroy(rn_object self);

rn_renderer RN_object_getRenderer(rn_object self);
void RN_object_setRenderer(rn_object self, rn_renderer r);

rn_physic_handler RN_object_getPhysicHandler(rn_object self);
void RN_object_setPhysicHandler(
        rn_object self, rn_physic_handler phandler);

void RN_object_setFullCollidable(rn_object self);
rn_collision_handler RN_object_getCollisionHandler(rn_object self);

void RN_object_move(rn_object self, rn_vec3 translation);
void RN_object_moveTo(rn_object self, rn_point3 location);
```

```
rn_box RN_object_getBB(rn_object self);
rn_box RN_object_getBoxPtr(rn_object self);

rn_point3 RN_object_getCenter(rn_object self);

void RN_object_rotateDegrees(rn_object self, float angle, rn_vector3 axis);

/*ObjectLoader*/
rn_texture_map rn_texture_map_create();
void rn_texture_map_destroy(rn_texture_map map);
rn_mesh RN_object_loader_loadMeshFromObj(const char *obj);
void RN_object_loader_obj_loadTextures(rn_texture_map map,
        const char* objStr, int qual);

/*Mesh*/
rn_box RN_mesh_getBB(rn_mesh self);

void RN_mesh_rotateDegrees(rn_mesh self, rn_vec3 axis, float ang);

void RN_mesh_scale(rn_mesh self, float sx, float sy, float sz);

/*MeshFactory*/
rn_mesh RN_mf_createBox(rn_vec3 dim);
rn_mesh RN_mf_createSphere(unsigned int precisionx, unsigned int precisiony, float radius)
    ;

/*PhysicHandler*/
rn_physic_handler RN_physic_handler_create(float mass);
void RN_physic_handler_destroy(rn_physic_handler self);

void RN_physic_handler_setBB(rn_physic_handler self, rn_box b);
void RN_physic_handler_addVolume(rn_physic_handler self, rn_volume vol);

void RN_physic_handler_setObjPhysicalProperty(
        rn_physic_handler self, ObjPhysicalProperty tp);

void RN_physic_handler_setGravity(rn_physic_handler self, rn_vec3 gVec);


rn_vec3 RN_physic_handler_getLinearVelocity(rn_physic_handler self);
void RN_physic_handler_setLinearVelocity(
        rn_physic_handler self, rn_vec3 velocity);

void RN_physic_handler_applyCentralForce(
        rn_physic_handler self, rn_vec3 force);

/*Point3*/
rn_vec3 RN_point3_difference(
        rn_point3 first, rn_point3 second);

float RN_point3_distance(
        rn_point3 first, rn_point3 second);

inline rn_point3 __to_rn_point3(const ronin::Point3 & p) {
```

```
    rn_point3 rnp;
    rnp.x = p.getX();
    rnp.y = p.getY();
    rnp.z = p.getZ();
    return rnp;
}

inline ronin::Point3 __to_ronin_point3(const rn_point3 & p) {
    return ronin::Point3(p.x, p.y, p.z);

/*Simulation*/
    rn_simulation RN_simulation_create(rn_simul_tp t);
void RN_simulation_addObject(
        rn_simulation self, rn_physic_handler obj);

void RN_simulation_removeObject(rn_simulation self, rn_physic_handler obj);

void RN_simulation_step(rn_simulation self, float step);

vector RN_simulation_getCollisions(rn_simulation self);

/*Vector3*/
inline rn_vector3 __to_rn_vector3(const ronin::Vector3 & p) {
    rn_vector3 rnp;
    rnp.x = p.getX();
    rnp.y = p.getY();
    rnp.z = p.getZ();
    return rnp;
}

inline ronin::Vector3 __to_ronin_vector3(const rn_vector3 & p) {
    return ronin::Vector3(p.x, p.y, p.z);
}

rn_vec3 RN_vec3_crossProduct(rn_vec3 first, rn_vec3 second);

float RN_vec3_abs(rn_vec3 self);

void RN_vec3_selfNormalize(rn_vec3* self);
void RN_vec3_selfScalarMultiply(
        rn_vec3* self, float scalar);
```

6.4 Artigo

CEx: Uma linguagem para programação concorrente e de tempo real com biblioteca para desenvolvimento de jogos

André Ferreira Bem Silva[1], **Felipe Borges Alves**[1], **Olinto José Varela Furtado**[1]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476 - Florianópolis/SC, Brasil

{dedefbs,fbafelipe,olinto}@inf.ufsc.br

Abstract. *This article describes a language that aims to ease the programming of soft real-time and concurrent systems. The defined language extends the standard C language, adding some new constructions to it. The language standard library provides some graphical utilities for game development, object file loading, rendering optimization passes, builtin physical simulation, high-level collision detection, 3D audio and shader loading. It also provides basic concurrent programming structures and real-time support by some functions which changes real-time exception handling. This extension, a.k.a as CEx, was implemented as a extension of the Clang compiler that generates code for the Low Level Virtual Machine (LLVM). From this implementation was possible to provide an environment for game development, which was tested by some simple prototypes.*

Resumo. *Esse artigo descreve a proposta de implementação de uma linguagem que visa facilitar a programação concorrente e de tempo real. A linguagem proposta estende a linguagem padrão C, adicionando algumas construções inexistentes nela. A biblioteca padrão da linguagem possui funcionalidades de programação de jogos que inclui primitivas de renderização, carregamento de arquivos de objetos, passos de otimização de renderização, simulação física, detecção de colisão, áudio 3D e carregamento de shaders. Também é função dessa biblioteca prover as estruturas básicas de programação concorrente e o suporte a programação de tempo real por meio de funções que alteram a captura de exceções de tempo real. Essa extensão, denominada CEx, foi implementada no compilador Clang, o qual gera código para a Low Level Virtual Machine (LLVM). A partir dessa implementação desenvolveu-se a um ambiente para desenvolvimento de jogos, o qual foi testado com alguns protótipos simples.*

1. Introdução

Pode-se observar uma demanda crescente por facilidades de abstração que permitam ao desenvolvedor escrever menos código, mantendo as capacidades de depuração e eficiência do código gerado. Isso não é diferente com programação paralela e tempo real. As ferramentas para programação de ambos tendem a ser adicionadas por bibliotecas, como nas linguagens C e C++. Algumas outras linguagens como Erlang[Armstrong 2003] e Concurrent C/C++[Peter et al. 1996] apresentam instruções da linguagem que promovem capacidades similares as fornecidas por biblioteca. Existem também as linguagens híbridas, que apresentam instruções e suporte na biblioteca para tempo real e concorrência.

Nessa classe é que encontra-se a linguagem CEx, fornecendo uma solução híbrida de modo a facilitar a programação nesses tipos de ambientes. Para implementar a linguagem, utilizamos da máquina virtual LLVM, associada ao frontend Clang.

A máquina virtual LLVM, possui um assembly próprio do tipo SSA (*Static Single Assignment*), o qual assemelha-se com os assemblys do tipo CISC. É possível gerar código para inúmeras máquinas reais a partir do código intermediário LLVM. O Clang é um projeto que pretende prover um frontend C/C++/Objective-C/Objective-C++ para a máquina virtual LLVM, substituindo assim o atual frontend mais comum do LLVM para essas linguagens que é o LLVM-GCC, baseado no conhecido compilador de código aberto. O Clang implementa o padrão C completo e gera código corretamente para essa linguagem.

2. Linguagem CEx

As seguintes construções foram definidas modificando a gramática C padrão[ISO 1999].

2.1. Novas estruturas de dados básicas

As novas estruturas de dados são TreeSet, HashSet, List, Vector, HashMap e TreeMap. A sintaxe e semântica para essas construções foram inspiradas nos templates de C++, os quais provêem na *Standard Template Library* da linguagem várias estruturas de dados básicas e genéricas.

```
<data_struct_type_specifier> ::= <type_specifier> | <pointer type_specifier> |
        CONST <pointer type_specifier>
<data_struct_type> ::= '<' <data_struct_type_specifier> '>'
<data_struct> ::= LIST <data_struct_type> | VECTOR <data_struct_type> |
        HASH_MAP <data_struct_type> | TREE_MAP <data_struct_type> |
        HASH_SET <data_struct_type> | TREE_SET <data_struct_type>
```

2.2. Construções de concorrência

Foram criadas cinco instruções de concorrência. As keywords atomic e synchronized servem para sincronização, enquanto as keywords paralel, parallel_for e async realizam paralelização automática de bloco (ou laço no caso do parallel_for).

2.2.1. atomic

Garante o acesso atômico a variáveis[1] de modo a tornar operações aritméticas, binárias, lógicas e de comparação atômicas, sempre que possível. O comportamento de atomic para operadores que não são unários e não fazem parte do conjunto multiplicação, divisão, soma e diferença é indefinido.

```
<type_qualifier> ::= ATOMIC
```

2.2.2. synchronized

Sincronização automática do statement alvo para acesso a ele com múltiplas threads de modo a tornar exclusividade o acesso ao bloco. Funciona para mutexes, conditions, barreiras e semáforos.

```
<synchronized_statement> ::= SYNCHRONIZED '(' <identifier_list> ')' <statement>
<statement>::= <synchronized_statement>
```

[1] Uma dentre {unsigned char, char, usigned shor, short, unsigned int, int, unsigned long long, long long}

2.2.3. parallel

A palavra chave parallel define que o bloco indicado será executado em paralelo, ou que obtenha informações relacionadas à execução do bloco paralelizado, ao contrário do async que indica um código cuja semântica estabelce independência de dados e nenhum tipo de troca de informação por meio da instrução.

```
<function_definition_async>::= PARALLEL <function_definition_sync>
<parallel_statement> ::= PARALLEL <statement>
<statement>::= <parallel_statement>
```

2.2.4. async

Modificador de funções e blocos de código, para executar em alguma thread específica. Indica que o bloco pode ser executado assincronamente, isto é, sem nenhuma relação com o bloco atual e portanto não há nenhum controle quanto ao conhecimento de término ou quanto ao estado da execução do bloco. Também não há nenhum tipo de troca de informação por meio da instrução, porém é possível obter-se informações referentes a execução do trabalho a partir da biblioteca padrão da linguagem.

```
<function_definition_async>::=
        SYNC '(' IDENTIFIER ')' <function_definition_sync>
<parallel_statement> ::= ASYNC '(' IDENTIFIER ')' <statement>
<statement>::= <parallel_statement>
```

2.2.5. parallel_for

Estrutura de repetição “for” onde as iterações são executadas em paralelo. Não há nenhum tipo de tratamento de dependência entre as iterações executadas em paralelo nesse caso.

```
<iteration_statement> ::=
        PARALLEL_FOR '(' TYPE_NAME IDENTIFIER ':' IDENTIFIER ')' <statement> |
        PARALLEL_FOR '(' TYPE_NAME IDENTIFIER ':' '[' CONSTANT ',' CONSTANT ']' ')' <statement> |
        PARALLEL_FOR '(' <expression_statement> <expression_statement> ')' <statement> |
        PARALLEL_FOR '(' <expression_statement> <expression_statement> <expression> ')' <statement>
```

2.3. Construções de tempo real

Para as instruções de tempo real, o suporte para o bloco aperiódicos é adicionado via *do_rt*, enquanto o suporte a blocos periódicos é feito via a instrução *periodic_rt*. Ambos indicam uma *deadline* em sua sintaxe e caso ela não seja obedecida uma exceção de tempo real é gerada e possivelmente tratada pelo programa.

2.3.1. do_rt

Essa instrução indica que o bloco alvo deve ser executado dentro de um tempo pré-estabelecido (*deadline*). Desse modo um bloco que seja indicado com essa palavra reservada deve executar com tempo máximo constante, conforme especificado. O comportamento padrão, caso a exceção de tempo real não seja tratada pela Task ou Thread executante, é de abortar o programa, funcionalidade similar a um sinal de *kill* do padrão POSIX, cuja especificação básica pode ser vista em [IEEE 2004].

```
<rt_statement> ::= DO_RT '(' <constant_expression> ')' <statement>
<statement> ::= <rt_statement>
```

2.3.2. periodic_rt

A semântica e sintaxe são idênticas ao do do_rt, a diferença está no fato que o bloco é tratado como periódico e deve ser executado antes do fim da *deadline* apontada, para cada iteração. Similar à utilizar-se um while com um do_rt como statement (similar porém não igual, uma vez que o while poderia executar mais de uma vez dentro da deadline, por exemplo). A primeira expressão constante refere-se ao tempo de deadline e a segunda à periodicidade do bloco.

```
<periodic_rt_statement> ::=
        PERIODIC_RT '(' <constant_expression> ')' '(' <constant_expression> ')' <statement>
<statement> ::= <periodic_rt_statement>
```

2.4. Geração de código

Para gerar assembly LLVM para as instruções referentes foram adotadas algumas diferentes estratégias diferentes para determinados grupos de instruções. As instruções podem ser divididas em três grupos descritos nas seções seguintes.

2.4.1. Instruções do_rt e synchronized

Para o do_rt e synchronized foram adotadas as mesmas estratégias, portanto eles são agrupados como grupo de solução nessa seção. No código llvm gerado, cria-se um prólogo e epílogo no bloco alvo do do_rt/synchronized. Em ambos, adiciona-se uma chamada para uma função com uma assinatura do tipo "__builtin_NomeDaKeyword_pre" e "__builtin_NomeDaKeyword_pos", para o prólogo e o epílogo (nessa sequência). No prólogo do do_rt, passam-se a expressão referente a *deadline*, enquanto no synchronized são passadas as variáveis utilizadas. No epílogo do do_rt passam-se o identificador da deadline, retornado pela função chamada no prólogo, enquanto no synchronized apenas as variáveis são passadas (na mesma sequência) para a função de liberação dos mecanismos de sincronia.

2.4.2. Instruções async, periodic_rt e parallel

Na implementação do async, periodic_rt e parallel foram adotadas o mesmo tipo de estratégia. Os blocos referentes a qualquer um dessas instruções são removidos do corpo da função e uma função especial é forjada (em tempo de compilação), acomodando o código contido. No lugar do corpo da função fica uma chamada para a biblioteca intermediária do tipo "__builtin_NomeDaKeyword_call", passando um ponteiro para a função forjada como parâmetro e os outros parâmetros específicos de cada instrução. O parallel_for também se encaixa nessa mesma categoria, mas sua estrutura especial será explicada na próximo seção.

2.4.3. Instrução parallel_for

Para implementar o parallel_for, também foi adotada a estratégia de inserir uma chamada no corpo da repetição e forjar uma função que é passada como parâmetro para a biblioteca. Porém, dada a complexidade maior, graças a estrutura ser iterativa, insere-se uma

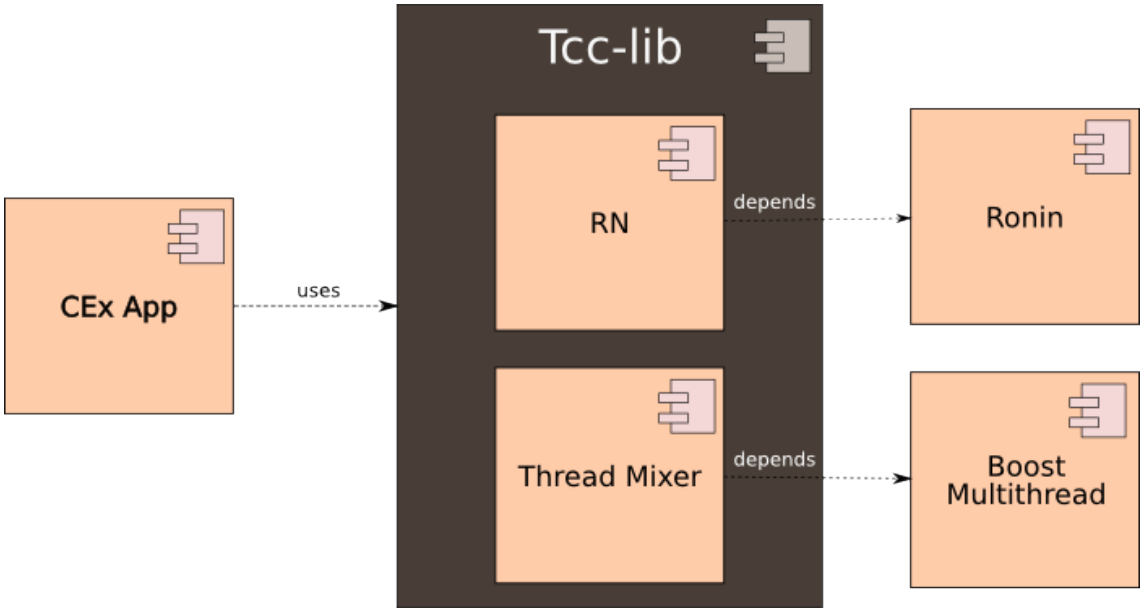

Figura 1. Arquitetura das bibliotecas CEx

chamada que cria uma estrutura de dados que será utilizada para armazenar as chamadas de cada iteração no prólogo do parallel_for. Para preencher essa estrutura, itera-se normalmente no *loop*, passando ela para a camada intermediária na chamada de função e por fim, ao acabar o *loop*, manda-se executar todas as tarefas encontradas. Naturalmente, a expressão de condição de iteração não deve possuir dependência de execução com o corpo da estrura de repetição.

3. Bibliotecas

Com o intuito de prover as facilidades específicas de paralelismo e primitivas para renderização gráfica, deve ser implementada uma biblioteca básica, suportada por algumas camadas de abstração, não visíveis diretamente para o programador. Dentre essas, destacam-se duas que são o motor gráfico Ronin e a Tcc-lib. Assim, a Tcc-lib fornece as funções de concorrência e tempo real, enquanto o motor gráfico Ronin providencia funções de renderização, colisão, etc. Um aplicativo qualquer CEx segue a arquitetura da figura 1.

3.1. Tcc-lib

A biblioteca padrão da linguagem, denominada Tcc-lib, consitui-se de duas partes: o Thread Mixer e a Ronin C Interface (RN).

3.1.1. Thread Mixer

É a parte da biblioteca que permite execução de várias *Tasks*, relacionadas em *Jobs*, que é um gerenciador de threads. O principal objetivo dessa camada é prover robustez e facilidade de paralelização de tarefas a serem executadas, abstraindo assim toda a parte de criação de Threads e utilização efetiva das unidades de processamento disponíveis. Essa parte da biblioteca utiliza extensivamente as estruturas de programação concorrente da biblioteca multithread do Boost C++ internamente (interface C).

Computador	Dual Core	Quad Core
Memória	4GB - 800mhz	4GB - 800mhz
Placa de vídeo	Geforce 9600M GT 512MB	Geforce 9600 GT 512 MB
Processador	Intel Core 2 Duo 2.13Ghz	AMD Phenom X4 2.0Ghz

Tabela 1. Configurações

Linguagem	CEx	C
Linhas de código	25	45

Tabela 2. Produtor-consumidor, CEx vs C

3.1.2. RN - Ronin C Interface

O suporte a renderização de primitivas gráficas e controle desses objetos primitivos faz parte das necessidades básicas de uma API de jogos, e para tanto, nenhuma solução já existente, individualmente, pareceu-nos completa e interessante o suficiente. Por isso e pelos motivos já descritos decidimos fazermos nossa própria engine, escrita em C++. Como a extensão foi proposta para C, fez-se necessário criar uma API C para a engine, chamada de RN.

3.2. Engine Ronin

A engine *Ronin*[Alves 2008], criada inicialmente por Felipe Borges Alves e codificada também por André Ferreira, tem suporte a programação de primitivas gráficas por meio do OpenGL (internamente).

Entre as características presentes na engine estão: Renderização por meio de objetos de alto nível, animações, texturas complexas/simples, detecção de colisão otimizada, view culling, carregamento de imagens em diversos formatos (.bmp, .png, .jpeg), carregamento de objetos complexos (.obj, .md2), sockets de alto nível, simulação física integrada com Bullet, a qual foi analisada em [Boeing and Bräunl 2007], carregamento de shaders, abstração de entrada (teclado, mouse, joystick, eventos de janela).

4. Exemplos clássicos e propostos

Para validar e testar as construções da linguagem utilizou-se alguns testes conhecidos e implementou-se mais alguns. Entre os testes clássicos estão o jantar dos filósofos, paralelização da sequência de Fibonacci e produtor-consumidor. Já no caso dos propostos encontram-se o conjunto de Mandelbrot, o teste de física da engine e um First Person Shooter (FPS).

Para todos os testes, foram utilizadas duas configurações distintas de computadores. Um dual core e outro quad core. A tabela 1 demonstra ambas as configurações.

4.1. Produtor-consumidor e jantar dos filósofos

A título de comparação implementou-se um exemplo de produtor-consumidor em CEx, e outro em C. Comparando-se assim o número de linhas totais de ambas as implementações pode-se observar que a implementação em CEx foi bem menos densa. Isso demonstra a

PC / Tipo - Threads	*Dual Core*	*Quad Core*
for - X	4.36s	3.26s
parallel - 1	4.29s	2.98s
parallel - 3	2.62s	1.54s
parallel - 6	2.58s	1.35s

Tabela 3. Tabela de tempos para término da execução dos laços

PC / Tipo de código	Dual Core	Quad Core
Código sequencial	9.5qps	7.43qps
Código paralelizado	18.12qps	19qps

Tabela 4. Performance de desenho, em quadros por segundo (qps)

maior facilidade, quando comparado a C, em escrever-se uma funcionalidade paralelizada. O comparativo pode ser observado na tabela 2.

4.2. Sequência de Fibonacci

O teste da sequência de Fibonacci visa testar a eficiência de paralelização de laços do CEx, relacionando a estrutura for ao parallel_for quanto ao desempenho. A tabela 3, demonstra os resultados para o nível O3 de otimização do Clang.

4.3. Conjunto de Mandelbrot

O conjunto de Mandelbrot é um exemplo de fractal, cujos tempo de renderização para versão paralelizada e sequencial podem ser observados na tabela 4. Pode-se observar nos resultados que a diferença de desempenho entre o dual core e o quad core foi pouca porque no caso do fractal, a renderização ocorre somente em uma thread, e portanto, a comunicação CPU com GPU acaba tornando-se o gargalo da aplicação.

5. Conclusão e trabalhos futuros

A partir da especificação das instruções foi possível criar produções na gramática básica do C e implementá-las com êxito em um compilador, no caso o Clang, dando a essas definições o nome de CEx.

Por meio da implementação realizada com o Clang, realizou-se vários testes que demonstram a eficácia geral das instruções rodando em vários ambientes diferentes. Nos testes foi possível observar o ganho de eficiência com a paralelização e também pode ser observado a utilização da extensão para as mais diversas aplicações e não somente para programação de jogos. A facilidade, em relação à bibliotecas padrões da linguagem, de codificar-se para programação concorrente foi demonstrada por meio de exemplos. Para melhoria da implementação feita, seria interessante complementar os trabalhos atuais. As possíveis melhorias futuras para a linguagem seriam:

- Implementação da gramática completa - falta atomic e as estruturas de dados
- Depuração avançado - tratamento de deadlocks
- C++Ex - modificações na gramática do C++

Entre as melhorias previstas para as próximas versões da engine Ronin encontram-se:

- Som 3D - suporte por meio de OpenAL, por exemplo
- Suporte a mapas - algum formato conhecido
- Efeitos gráficos pós-processados - fogo, água, vento, refração...
- Renderizador multithread - melhoria da implementação atual
- Módulo de IA - criar API

Referências

Alves, F. B. (2008). Engine ronin.

Armstrong, J. (2003). Making reliable distributed systems in the presence of software errors.

Boeing, A. and Bräunl, T. (2007). Evaluation of real-time physics simulation systems. In *GRAPHITE '07: Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, pages 281–288, New York, NY, USA. ACM.

IEEE (2004). Standard for information technology - portable operating system interface (posix). shell and utilities. Technical report.

ISO (1999). Iso c standard 1999. Technical report. ISO/IEC 9899:1999 draft.

Peter, P., Buhr, A., and Sartipi, K. (1996). Concurrent c/c++ programming languages.