Módulo de Gerenciamento de Memória para um Framework de Desenvolvimento de Jogos

Mateus dos S. Garcia[1], **Francisco Pereira Junior**[1], **Henrique Yoshikazu Shishido**[1], **Rogério Santos Pozza**[1]

[1]Coordenação de Análise e Desenvolvimento de Sistemas
Universidade Tecnológica Federal do Paraná (UTFPR)
Av. Alberto Carazzai, 1640 – 86.300-000 – Cornélio Procópio – PR – Brazil

matgar@gmail.com, {fpereira, shishido, pozza}@utfpr.edu.br

Abstract. *This paper presents the development of the memory management system for a game development framework titled Crazy Framework. The objective of this system is to manage the memory used by applications created from this framework, providing information about the allocated blocks and usage statistics to developers, especially useful to avoid common problems associated with the chaotic use of the heap region, including: fragmentation, dangling pointers and memory leaks.*

Resumo. *Esse artigo apresenta o desenvolvimento do módulo de gerenciamento de memória para um framework de desenvolvimento de jogos eletrônicos intitulado Crazy Framework. O objetivo do módulo é gerenciar a memória utilizada por aplicações criadas a partir do framework, oferecendo aos desenvolvedores informações sobre os blocos alocados e estatísticas de utilização, úteis principalmente para evitar problemas comuns, relacionados com a utilização caótica da região da heap, entre eles: fragmentação, dangling pointers e memory leaks.*

1. Introdução

Em [Chen 2004] *frameworks* são definidos como um conjunto de *designs* e códigos reusáveis que podem auxiliar o desenvolvimento de aplicações de software. Os *frameworks* fornecem aos desenvolvedores uma estrutura baseada em classes abstratas, classes concretas e interações pré-definidas entre elas. Esse artigo tem como objetivo apresentar o módulo de gerenciamento de memória para um *framework* de desenvolvimento de jogos eletrônicos intitulado *Crazy Framework*. Ressalta-se que este é o primeiro e único módulo criado até o momento.

Foram criadas interfaces abstratas para o módulo que agregam funcionalidades para tornar mais eficiente o controle da memória em novas aplicações, facilitam sua utilização por parte dos desenvolvedores e permitem sua implementação em diferentes plataformas de hardware. Por meio dessas interfaces é possível criar uma visualização do estado da memória em determinado momento, com dados sobre para qual módulo, recurso ou grupo de funcionalidade (texturas, geometrias, arquivos, etc.) os blocos foram alocados e o tamanho de cada um deles.

Para verificar as vantagens e a viabilidade de se construir um *framework*, foram realizadas pesquisas sobre as funcionalidades de ferramentas de desenvolvimento

disponíveis no mercado e sobre diferentes soluções relacionadas com o gerenciamento de memória propostos por diferentes autores.

Esse artigo possui cinco seções além desta: a seção 2 apresenta a revisão bibliográfica com diferentes ferramentas e *frameworks* voltados para o desenvolvimento de *games*; a seção 3 descreve problemas relacionados com a utilização de memória e a abordagem de gerenciamento utilizada no módulo; a seção 4 descreve a metodologia de testes para validar a performance dos algoritmos de gerenciamento de memória implementados; a seção 5 apresenta uma análise dos resultados obtidos com os testes; e a seção 6 traz as considerações finais e trabalhos futuros.

2. Revisão Bibliográfica

Algumas tecnologias do mercado são destinadas a manipulação de diferentes recursos e funcionalidades de hardware, como a biblioteca *Open Graphics Library* (OpenGL) e o *DirectX* da *Microsoft*, ou abranger todos os aspectos do desenvolvimento, como as *game engines*.

Em [Woo et al. 1997] o OpenGL é descrito como uma interface de software (implementada em diferentes plataformas de hardware) composta por cerca de 150 comandos usados para especificar operações necessárias para produzir aplicações em 3D, podendo construir modelos complexos apenas com primitivas geométricas (pontos, linhas e polígonos).

O *DirectX* é um *Software Development Kit* (SDK) composto por uma coleção de bibliotecas *Component Object Model* (COM) [Thorn 2005]. Entre suas bibliotecas encontramos: *Direct3D* (gráficos), *DirectShow* (manipulação de arquivos de mídia), *DirectInput* (entrada de dados por meio de teclado e mouse), *DirectSound* (sons) e *DirectPlay* (comunicação via rede).

Apesar das funcionalidades, o OpenGL e o *DirectX* se caracterizam como "caixas de ferramentas". Não incorporam aspectos como, por exemplo, inteligência artificial, simulações de física (como colisões e gravidade), tratamento de eventos, flexibilidade e extensibilidade das suas capacidades.

Em contrapartida, *engines* são ferramentas para desenvolvimento de jogos cujo objetivo é fornecer algumas funcionalidades ao projeto do software [Zerbst 2004]. A *CryEngine 3*[1], por exemplo, oferece um conjunto de ferramentas para facilitar o desenvolvimento de aplicações. Porém, os preços de suas licenças e os *royalties* sobre as vendas dos produtos produzidos, normalmente são inviáveis para pequenas empresas e desenvolvedores independentes.

Frameworks por sua vez, são aplicações reusáveis e semi-completas que podem ser especializadas para produzir novas aplicações customizadas [Fayad and Schmidt 1997]. Sua estrutura é representada por classes abstratas e a interação de suas instâncias (base da customização). Suas interfaces com alto nível de abstração tornam os sistemas resultantes fáceis de manter, confiáveis e eficientes.

A Tabela 1 apresenta o comparativo entre três *frameworks* muito utilizados para o

[1] http://www.crytek.com

desenvolvimento de *games*: *XNA's Not Acronymed*(XNA)[2] da *Microsoft*, *SexyAppFramework*[3] da *PopCap* e o *Homura Games Framework*[4], desenvolvido em *Liverpool John Moores University*, na Inglaterra. Desta forma é possível apontar suas funcionalidades, confrontar as tecnologias empregadas e destacar as particularidades que cada um apresenta, e consequentemente guiar um novo desenvolvimento.

Tabela 1. Comparativo entre *frameworks* de desenvolvimento de *games*

	XNA Game Studio	*SexyAppFramework*	*Homura*
Gráficos	*DirectX* 2D e 3D	2D e 2D acelerado via *DirectX*	OpenGL 2D e 3D
Matemática	Manipulação de vetores, matrizes e detecção de colisões	Vetores e matrizes para manipulação 2D	Biblioteca matemática baseada nas funcionalidades do *Java Monkey Engine*
Física	-	-	Biblioteca de simulação de física baseada na ODE
Áudio	Funcionalidades de áudio próprias e simulação de sons em 3D	Utilização da biblioteca de áudio BASS, podendo executar os formatos de arquivos MP3 e OGG	Utilização da biblioteca de áudio OpenAL para execução de efeitos sonoros e músicas
Mídia	*Content Pipeline* responsável por carregar arquivos de mídia para a aplicação.	*Resource Manager* configurável em arquivos XML, carrega sons e imagens para a aplicação.	Gerenciador de conteúdos extensível capaz de abrir arquivos compactados JAR e PAK.
Plataformas	Computadores com sistema operacional *Windows* e o console *Xbox 360*	Computadores com sistema operacional *Windows*	Plataforma *Web*
Gerenciamento de Memória	Funções de gerenciamento de memória do CLR	Funções de gerenciamento próprias baseadas em mapeamento de memória	Funções de gerenciamento de memória automática por meio do JVM.

Dois dos *frameworks* comparados (*XNA* e *Homura*) são escritos em linguagens de alto nível que possuem mecanismos próprios para o gerenciamento de memória da aplicação (*garbage collector*). O *SexyAppFramework*, que assim como o *Crazy Framework* é implementado em C/C++, possui funcionalidades próprias para realizar o gerenciamento.

Tomando como base as funcionalidades disponíveis na biblioteca OpenGL e no *DirectX*, as condições de utilização das *engines* apresentadas e os três *frameworks* comparados, torna-se necessário a criação de uma ferramenta própria que ofereça um mínimo de funcionalidades para iniciar a produção de jogos em uma empresa. Além disso, em

[2]http://msdn.microsoft.com/en-us/library/bb200104.aspx
[3]http://forum.fischeronline.de/
[4]http://java.cms.livjm.ac.uk/homura/

[Jones and Lins 1996] e [Marquet 2007], estimou-se que o gerenciamento de memória poupa cerca de 40% do tempo de desenvolvimento de uma aplicação, deixando clara a necessidade dessa funcionalidade na ferramenta a ser criada.

3. Gerenciamento de Memória no Crazy Framework

Como apresentado em [Llopis 2003], a *heap* é uma região da memória do processo capaz de acomodar objetos não temporários ou limitados a um escopo, mas por causa dessa característica sofre vários problemas ao decorrer de sua utilização. Por não possuir ordem fixa de alocação e liberação de blocos de memória, a *heap* pode ser a fonte de problemas como: fragmentação de memória, *dangling pointers*, *memory leaks* entre outros.

A fragmentação de memória ocorre quando um espaço de memória sofre diversas alocações e liberações a ponto de não ser possível alocar novos blocos, mesmo com quantidade de memória disponível entre os blocos já alocados. Os *dangling pointers* são ponteiros que referenciam endereços de memória que não estão sendo mais usados, ou que tiveram seu conteúdo inadivertidamente alterado por outra parte da aplicação. Quando esses dados inválidos são manipulados, podem levar a aplicação a parar sua execução. Já os *memory leaks* são blocos de memória que não estão sendo mais referenciados pela aplicação e não foram devolvidos ao sistema, fazendo com que o consumo de memória suba gradativamente até que o sistema fique sem memória.

Sistemas Operacionais (SO) modernos implementam técnicas de gerenciamento de memória (compactação e memória virtual, por exemplo) para que esses problemas não interfiram na execução de suas tarefas. Mas no contexto da aplicação, a *heap* pode ser um espaço de memória de tamanho definido, pré-alocado junto ao SO, também conhecido como *pool* de memória. Desse modo, podem ocorrer problemas como a fragmentação mesmo com a gerência do SO. Para soluciná-los, é necessário criar métodos e regras que governem as alocações nesse espaço.

O módulo de gerenciamento de memória do *Crazy Framework* é capaz de realizar as seguintes ações na região da *heap*: alocar e liberar memória, listar as alocações e consultar estatísticas de utilização de memória. Para isso, conta com duas classes principais, a *CCrazyAlocação* e a *CCrazyHeap*. Na Figura 1 estão presentes as classes que compõem o módulo de gerenciamento de memória e as relações entre elas (para facilitar a visualização os métodos foram omitidos).

CCrazyAlocacao representa uma alocação feita na memória. *CCrazyHeap* é uma classe abstrata que representa um espaço dentro da memória, utilizada para criar alocações. Possui métodos extensíveis para alocação e liberação de memória, validação de alocações (corrompimento) e listagem dos blocos ativos na memória. Em uma aplicação podem coexistir várias instâncias de classes concretas, especializadas a partir da classe *CCrazyHeap*. Assim, cada uma dessas instâncias fica responsável por um tipo ou propósito de alocação na memória. As classes *CCrazyHeapGenerico*, *CCrazyHeapWindows* e *CCrazyPool* são implementações concretas da classe *CCrazyHeap*.

A *CCrazyHeapGenerico* implementa uma lista duplamente encadeada de alocações por meio de uma especialização da classe de alocações chamada *CCrazyAlocacaoGenericaInfo* e uma organização específica das informações do bloco da memória, contando com um prefixo e um sufixo utilizado para validação da integridade do bloco.

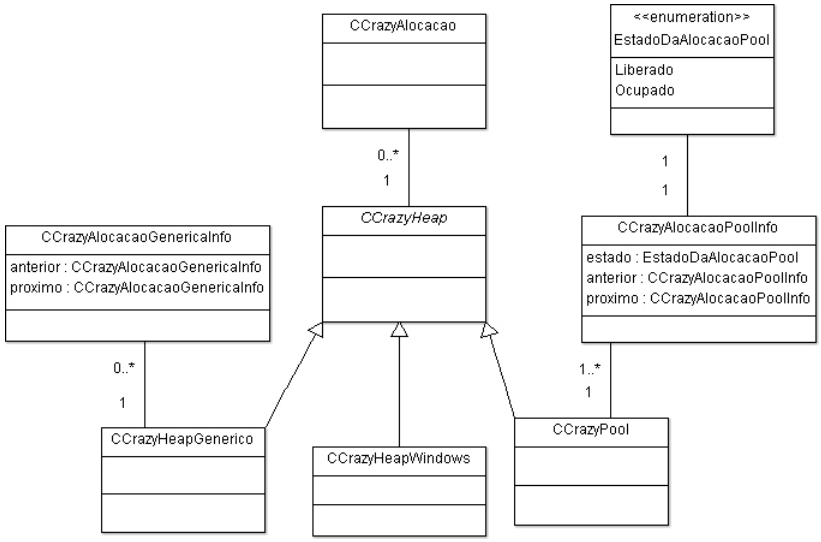

Figura 1. Diagrama de classes do módulo de gerenciamento de memória

O conceito de *pool* mencionado anteriormente é implementado pela classe *CCrazyPool*. Para gerenciar os blocos de memória, utiliza *sequential fit*, definido em [Blunden 2002] como uma técnica que organiza a memória em uma lista encadeada de blocos livres e ocupados. Nesse caso a especialização da classe de alocações *CCrazyAlocacaoPoolInfo* armazena esses dois estados do bloco.

Por último, a classe *CCrazyHeapWindows* encapsula as funcionalidades de gerenciamento de *heaps* implementadas pelo sistema operacional *Windows*, acessíveis pelo arquivo *windows.h*. As funcionalidades de gerenciamento do *Windows* utilizadas nessa classe foram: criação e destruição de *heaps*; alocação e liberação de memória para um *heap*; listagem de blocos alocados em uma *heap* e validação de blocos.

4. Testes

Para testar a performance de um gerenciador de memória é necessário executar uma série de alocações e liberações de memória, sendo que os blocos alocados não podem ser todos do mesmo tamanho. Por outro lado as alocações não podem ser totalmente aleatórias. Em [Blunden 2002] é proposto um algoritmo que cria um conjunto de alocações que seguem uma distribuição baseada em uma probabilidade discreta específica, permitindo atribuir um peso a certos tipos de pedidos de alocações, como mostra a Figura 2.

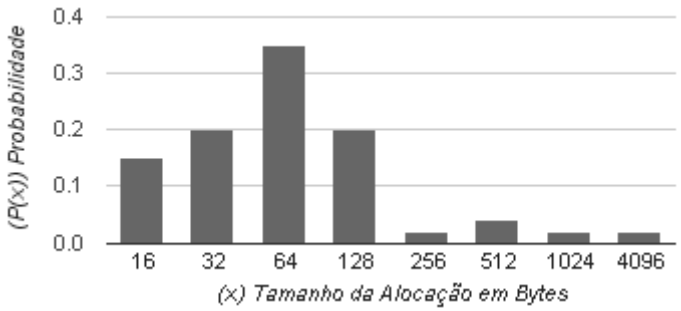

Figura 2. Probabilidade de alocações durante os testes

Para gerar números aleatórios que obedeçam essa distribuição é utilizado um algoritmo conhecido como *inverse transform method*. Essa abordagem, segundo [Blunden 2002], é sintética se comparada com outras opções disponíveis no mercado, mas para testar a performance dos algoritmos aqui implementados ela foi considerada adequada. Os passos do algoritmo são:

- Passo 1. Gerar um número aleatório, U, entre 0 e 1;
- Passo 2. Se U < p1 = 0.15, o tamanho da alocação é definido para 16 *bytes*, avançar para o passo 10;
- Passo 3. Se U < p1 + p2 = 0.35, o tamanho da alocação é definido para 32 *bytes*, avançar para o passo 10;
- Passo 4. Se U < p1 + p2 + p3 = 0.70, o tamanho da alocação é definido para 64 *bytes*, avançar para o passo 10;
- Passo 5. Se U < p1 + p2 + p3 + p4 = 0.90, o tamanho da alocação é definido para 128 *bytes*, avançar para o passo 10;
- Passo 6. Se U < p1 + p2 + p3 + p4 + p5 = 0.92, o tamanho da alocação é definido para 256 *bytes*, avançar para o passo 10;
- Passo 7. Se U < p1 + p2 + p3 + p4 + p5 + p6 = 0.96, o tamanho da alocação é definido para 512 *bytes*, avançar para o passo 10;
- Passo 8. Se U < p1 + p2 + p3 + p4 + p5 + p6 + p7 = 0.98, o tamanho da alocação é definido para 1024 *bytes*, avançar para o passo 10;
- Passo 9. O tamanho da alocação é definido para 4096 *bytes*, avançar para o passo 10;
- Passo 10. Parar a execução.

Na seção seguinte estão descritas as análises a cerca dos resultados obtidos pelo módulo implementado.

5. Análise e Discussão

Nos testes foram alocadas e em seguida liberadas, cada uma das quantidades de blocos de memória presentes nas abscissas dos gráficos das Figuras 3 e 4, sendo medido o tempo entre o início da alocação e o fim da liberação de cada um. O número de alocações foi sintético para forçar os algoritmos a mostrarem as suas diferenças de performance. Durante a execução de um *game*, milhões de alocações são realizadas. Mas em um único *frame*, muitas vezes o número de alocações é menor do que o número máximo apresentado nos gráficos de performance.

A Figura 3 apresenta os tempos de execução dos testes. Entre 1024 e 4096 a diferença entre os tempos de alocação não foram significativas. Entre 8192 e 65536 alocações, o aumento médio do tempo de execução, em milissegundos, dos algoritmos implementados é igual a 422,25 ms para a classe *CCrazyHeapGenerico* e 160 ms para o *CCrazyHeapWindows* em relação ao C/C++.

Na Figura 4 é possível verificar que a classe *CCrazyPool* apresentou tempos de execução até 593% maiores que as outras implementações, principalmente nos totais de alocações acima de 2048, chegando a 437 ms. Esse tempo está relacionado com a busca linear feita na lista de blocos do *pool*. Quanto maior o número de blocos, maior o tempo de busca por blocos que estejam livres e que tenham o tamanho requisitado pela alocação.

Os tempos de execução dos algoritmos apresentados são maiores do que a implementação padrão da linguagem C/C++ devido as abordagens utilizadas e o *overhead*

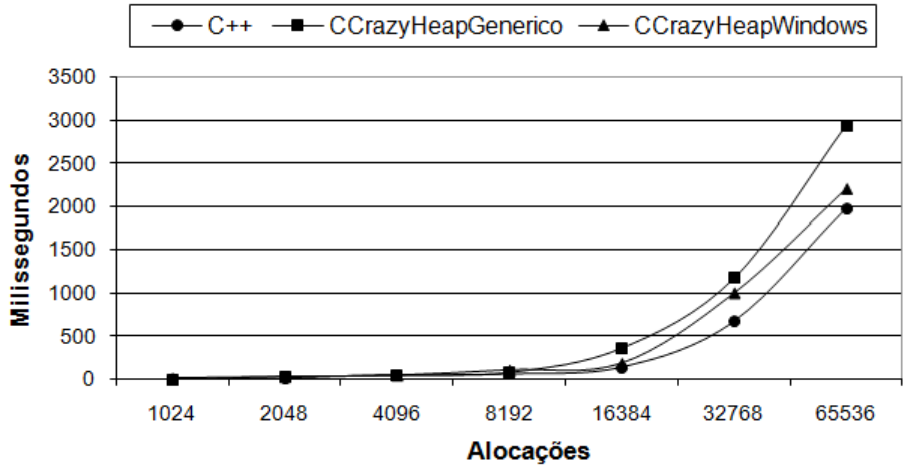

Figura 3. Desempenho das classes implementadas

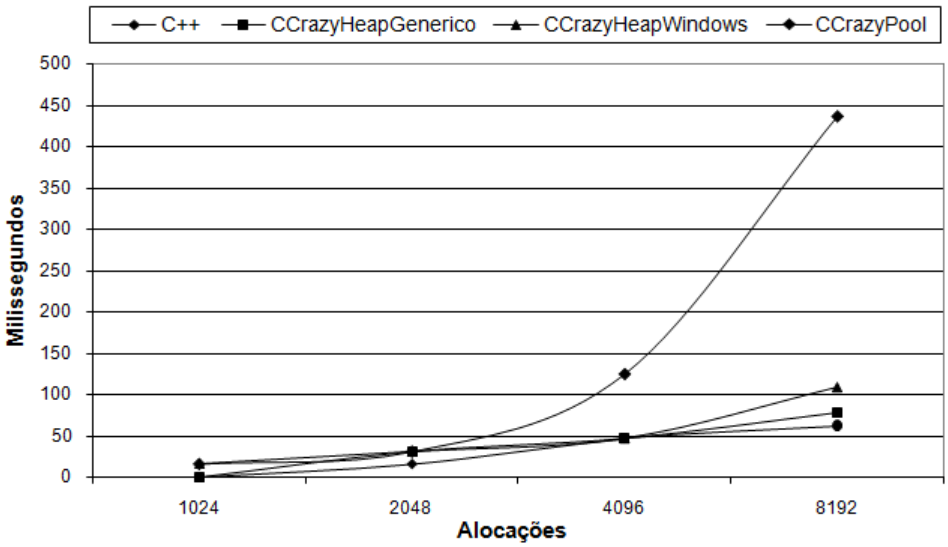

Figura 4. Desempenho da classe *CCrazyPool*

causado pelos dados auxiliares necessários em cada bloco alocado. Essa quantidade pode chegar a 112 *bytes* (no caso da classe *CCrazyHeapGenerico*) adicionais para qualquer tamanho de bloco alocado. Esses dados auxiliares são essenciais para otimizar o gerenciamento de memória e, como é possível observar na Figura 3, quanto menor a quantidade de alocações, mais próximos estão os tempos de execução das implementação deste módulo com a abordagem de alocação nativa do C/C++. Como dito no início dessa seção, o número de alocações em um único *frame* do jogo muitas vezes é menor do que o número máximo apresentado nos gráficos. Assim, a performance do módulo foi considerada aceitável para utilização em um *game*.

Como ainda não existem outros módulos construídos do *Crazy Framework* não é possível gerar um *game* para demonstrar as funcionalidades de listagens de blocos alocados do módulo de gerenciamento de memória. Porém, como o módulo sobrecarrega as funções *malloc* e *free* e os operadores *new* e *delete*, é capaz de gerenciar a memória de aplicações escritas em C/C++, sem que essas sejam relacionadas com o *framework*.

Em [Buckland 2005], são apresentados diversos conceitos relacionados com Inteligência Artificial (IA) voltada para o desenvolvimento de jogos eletrônicos. Ao fim dos estudos é apresentada uma aplicação chamada *Raven*[5], que implementa todos os assuntos abordados durante a obra. O *Raven*, é um *game* 2D de código aberto que foi utilizado no presente trabalho como ambiente de testes das funcionalidades do módulo de gerenciamento de memória do *Crazy Framework*. Foram definidos três grupos de objetos dentro da arquitetura do jogo: objetos do mapa, armamentos e inteligência artificial.

```
+-------------------------------------------------------------------------+
|                        Alocacoes do CCrazyHeap                          |
+-------------------------------------------------------------------------+
                      Data: Tue Dec 07 01:18:35 2010

+-------------------------------------------------------------------------+

Total        Ativas        Pico        Total de Bytes        Pico de Bytes
---------------------------------------------------------------------------
447          66            73                    3.1K                 3.5K
---------------------------------------------------------------------------
Numero   Endereco Tamanho Operador    Arquivo do Codigo Fonte
---------------------------------------------------------------------------
     0 0x00AF94DC    272B         New Raven_Game.cpp(257)
     1 0x00AF8814     32B         New Raven_Bot.cpp(64)
     2 0x00AF96AC    184B         New Raven_Bot.cpp(67)
     3 0x00AF9644     16B         New Raven_Bot.cpp(70)
     4 0x00AF97BC     16B         New Raven_Bot.cpp(71)
     5 0x00AF9824     16B         New Raven_Bot.cpp(72)
     6 0x00AF988C     16B         New Raven_Bot.cpp(73)
     7 0x00AF98F4     16B         New Raven_Bot.cpp(74)
     8 0x003B9754     44B         New Raven_Bot.cpp(77)
     9 0x00AF99C4     16B         New Goal_Think.cpp(42)
    10 0x00AF9A2C     16B         New Goal_Think.cpp(43)
    11 0x00AF9A94     16B         New Goal_Think.cpp(44)
    12 0x003BBAF4     24B         New Goal_Think.cpp(45)
    13 0x00AF9BCC     24B         New Goal_Think.cpp(47)
    14 0x00AF9CAC     24B         New Goal_Think.cpp(49)
    15 0x003BBB64      8B         New Raven_Bot.cpp(80)
    16 0x00AF9D1C     48B         New Raven_Bot.cpp(82)
    17 0x00CCEE9C     24B         New Raven_Bot.cpp(87)
    18 0x00B90094    272B         New Raven_Game.cpp(257)
    19 0x00B901FC     32B         New Raven_Bot.cpp(64)
    20 0x00B90394    184B         New Raven_Bot.cpp(67)
    21 0x00CCEF7C     16B         New Raven_Bot.cpp(70)
    22 0x00CCEFE4     16B         New Raven_Bot.cpp(71)
```

Figura 5. Lista de alocações do grupo de IA do *Raven*

O primeiro grupo representa os objetos do mapa que são carregados de uma única vez no início do jogo e destruídos também de uma única vez no seu término, assim uma instância da *CCrazyPool* foi utilizada para gerenciar suas alocações. O segundo grupo representa as armas e projéteis disparados pelo mundo, como são objetos que são instanciados e destruídos a todo momento durante a execução, foi utilizada uma instância da classe *CCrazyHeapGenerico* para gerenciar as alocações do grupo. O terceiro e último grupo representa os objetos de IA. Novamente foi utilizado o *CCrazyHeapGenerico* para gerenciar as alocações, já que os objetos desse grupo também são instanciados e destruídos a todo momento durante a execução do jogo. Objetos não gerenciáveis como os nós das listas dinâmicas da biblioteca padrão do C/C++ são alocados em uma *heap* padrão do tipo *CCrazyHeapGenerico*.

A Figura 5 mostra a listagem de alocações para o grupo de IA do jogo *Raven*. No cabeçalho está presente o título da listagem e a data em que foi gerada. Nas colunas: o total de alocações realizadas pelo *heap*, o total de alocações ativas no momento em que foi gerado o relatório, o pico de alocações atingido, o total de *bytes* ativos, e o pico de

[5]Jogo didático criado passo a passo em [Buckland 2005].

bytes que foram alocados. Logo abaixo estão apresentados: o número de cada alocação dentro do *heap*, o endereço de cada alocação, o tamanho total de cada um dos blocos, o operador utilizado na alocação e o arquivo do código-fonte onde o bloco foi criado.

No fim dos testes de execução do *Raven* as listagens não apresentaram *memory leaks*. O jogo foi executado durante 1 minuto, esperando que ações como a morte de um agente e o consumo de itens fossem realizadas. O total de memória utilizada no período de execução foi: 4.3 *Kbytes* com objetos do mapa, 656 *bytes* com armamentos e 3.5 *Kbytes* com objetos de IA. Esses valores e as informações associadas aos blocos demonstram que, com os relatórios, é possível não só encontrar problemas com a alocação de blocos mas também calcular o total de memória utilizado pela aplicação durante sua execução, fator importante quando se trabalha com hardwares com tamanho de memória limitado.

6. Considerações Finais

Os objetivos propostos nesse artigo foram alcançados e no geral os resultados obtidos foram satisfatórios. A classe *CCrazyHeap* define as principais características de uma *heap* dentro do contexto do *Crazy Framework* e seus métodos extensíveis permitem que desenvolvedores criem suas próprias soluções, dependendo da aplicação construída e das plataformas de hardware em que serão executadas. Uma *heap* pode ser instanciada e utilizada por um módulo ou conjunto de funcionalidades e múltiplas instâncias podem coexistir no ambiente da aplicação, melhorando assim a organização e o controle das alocações. Em contrapartida, por ser permitida essa coexistência, não é possível garantir que os blocos pertencentes a uma *heap* façam parte de uma região contígua de memória, fazendo da fragmentação um problema ainda à ser tratado. A integridade de cada bloco é realizada no momento de sua liberação, e uma lista das alocações ativas pode ser gerada para cada *heap*, auxiliando na verificação de *memory leaks*.

Como visto nos gráficos apresentados na seção 5, o acréscimo no tempo de execução das classes especializadas: *CCrazyHeapGenerico* e *CCrazyHeapWindows*, pode chegar a ser maior que 100%, em alguns casos, se comparado com a implementação padrão do C/C++. Essa diferença de performance se deve às informações adicionais alocadas junto ao novo bloco e a atualização das estatísticas de utilização, sendo assim justificada pela importância dessas informações.

A classe *CCrazyPool* apresentou maior tempo de execução de seus algoritmos, chegando à 593% de acréscimo nas alocações acima de 2048 objetos, tendo assim a pior performance. Isso se deve principalmente a busca sequencial por blocos livres registrados na lista dinâmica duplamente encadeada utilizada pelo *sequential fit*. Uma maneira de sanar esse problema seria dividir as alocações do *pool* em duas listas dinâmicas duplamente encadeadas, uma contendo os blocos ocupados e outra contendo os livres, reduzindo o tempo de busca do algoritmo.

Não foi mensurado o tempo de desenvolvimento que seria economizado utilizando o módulo de gerenciamento de memória do *Crazy Framework* em um novo projeto de jogo. Mesmo assim, as diversas interfaces abstratas, podem poupá-lo durante o desenvolvimento, não sendo necessária a reconstrução de partes do código, uma vez que são compartilhadas entre diferentes plataformas de hardware e sistemas operacionais. Como trabalho futuro é interessante mensurar este tempo, utilizando o módulo de gerenciamento de memória e compará-lo com o tempo de desenvolvimento utilizando outras ferramentas.

Com essa informação é possível analisar os benefícios que o módulo traz ao desenvolvimento do projeto.

Outros esforços poderiam ser direcionados na construção de uma ferramenta visual para ajudar o desenvolvedor a monitorar o estado da memória e as demais estatísticas fornecidas pelo módulo por meio de gráficos atualizados em tempo real durante a execução da aplicação. Esses dados poderão ser salvos, representando o conteúdo da memória em determinado momento. É interessante também a integração de técnicas de *garbage collector* para automatizar o gerenciamento no que diz respeito à liberação de memória para o sistema. Essas técnicas podem ajudar a sanar os *dangling pointers* e os *memory leaks* mencionados no trabalho e se construídas de maneira a aperfeiçoar o desempenho de execução dos algoritmos de coleta, podem trazer grandes benefícios as aplicações criadas a partir do *framework*.

De maneira geral, o módulo de gerenciamento de memória construído para o *Crazy Framework* fornece informações suficientes e métodos para minimizar os problemas relacionados com a utilização da memória em linguagens como o C/C++, auxiliando na construção de novos *games* a partir deste *framework*.

Referências

Blunden, B. (2002). *Memory Management: Algorithms and Implementations in C/C++*. Jones & Bartlett Publishers, Texas.

Buckland, M. (2005). *Programming Game AI By Example*. Wordware Publishing, Inc., Plano, Texas.

Chen, X. (2004). *Developing Application Frameworks in .NET*. APress, New York, NY.

Fayad, M. and Schmidt, D. C. (1997). Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38.

Jones, R. and Lins, R. (1996). *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, New York, NY.

Llopis, N. (2003). *C++ for Game Programmers*. Charles River Media, Inc., Hingham, Massachussets.

Marquet, K. (2007). *Gestion de mémoire à objets pour systèmes embarqués*. PhD thesis, Université des Sciences et Technologies de Lille, France.

Thorn, A. (2005). *DirectX 9 Graphics: The Definitive Guide to Direct3D*. Wordware Publishing, Plano, Texas.

Woo, M., Neider, J., and Davis, T. (1997). *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison-Wesley Professional, USA, 2 edition.

Zerbst, S. (2004). *3D Game Engine Programming*. Course Technology PTR, Boston, MA.