UNIVERSIDADE DO EXTREMO SUL CATARINENSE – UNESC
CURSO DE CIÊNCIA DA COMPUTAÇÃO

JONAS GABRIEL DE SOUZA

DETECÇÃO DE COLISÃO EM AMBIENTES BIDIMENSIONAIS E TRIDIMENSIONAIS UTILIZANDO ESTRUTURAS DE DADOS HIERÁRQUICAS QUADTREES E OCTREES

CRICIÚMA
2012

JONAS GABRIEL DE SOUZA

DETECÇÃO DE COLISÃO EM AMBIENTES BIDIMENSIONAIS E TRIDIMENSIONAIS UTILIZANDO ESTRUTURAS DE DADOS HIERÁRQUICAS QUADTRESS E OCTREES

Trabalho de Conclusão de Curso, apresentado para obtenção do grau de Bacharel no curso de Ciência da Computação da Universidade do Extremo Sul Catarinense, UNESC.

Orientador: MEng. Evânio Ramos Nicoleit

CRICIÚMA

2012

JONAS GABRIEL DE SOUZA

DETECÇÃO DE COLISÃO EM AMBIENTES BIDIMENSIONAIS E TRIDIMENSIONAIS UTILIZANDO ESTRUTURAS DE DADOS HIERÁRQUICAS QUADTREES E OCTREES

Trabalho de Conclusão de Curso aprovado pela Banca Examinadora para obtenção do Grau de Bacharel, no Curso de Ciência da Computação da Universidade do Extremo Sul Catarinense, UNESC, com Linha de Pesquisa em Computação Gráfica.

Criciúma, 29 de Junho de 2012.

BANCA EXAMINADORA

Prof. MEng. Evânio Ramos Nicoleit - UNESC - Orientador

Prof. MSc. Paracelso de Oliveira Caldas - UNESC

Profª. MSc. Leila Laís Gonçalves - UNESC

À Nietzsche, por ter matado deus.

AGRADECIMENTOS

Gostaria agradecer à minha família de um modo geral e, embora eu não tenha contato com todos, sei que estão comigo em pensamento.

Agradeço, também, a todos os professores e colegas que me acompanharam e que me apoiaram durante a minha jornada ao longo do curso.

Aos professores Evânio Ramos Nicoleit e Leila Laís Gonçalves, minha sincera gratidão. Nestes últimos quatro anos de convivência muito me ajudaram a crescer tanto profissionalmente como pessoalmente.

Não posso deixar de agradecer a todos os meus inúmeros amigos, em especial ao Gustavo Bueno, pois ele ficaria muito bravo se eu não o fizesse.

E sobretudo, agradecer ao Igor que, no meu momento de desespero, me incentivou a não desistir.

“Faz o que tu queres. Há de ser tudo da Lei.
O amor é a Lei, o Amor sob a Vontade”

Aleister Crowley

RESUMO

A detecção de colisão entre objetos e outros eventos situados no espaço é uma grande problemática nas áreas da simulação, uma vez que o custo computacional cresce de maneira proporcional à complexidade computacional envolvida. Portanto, a escolha da técnica utilizada para realizar estas tarefas se torna importante, tanto em questão de qualidade quanto de desempenho. Entretanto, é possível encontrar este equilíbrio entre robustez e eficiência com a utilização de quadtrees e octrees. Estas estruturas de dados hierárquicas, combinam desempenho e qualidade e podem atender à diversas necessidades, tornando-se uma solução flexível e ao mesmo tempo eficiente para a indexação de eventos no espaço. De modo a evidenciar a eficiência destas estruturas de dados, este trabalho aborda um estudo comparativo realizado entre as mesmas e o modo convencional de detecção de eventos no espaço com foco na detecção de colisão e em uma análise quantitativa dos resultados obtidos.

Palavras-chave: Detecção de colisão. Quadtree. Octree. Computação Gráfica.

ABSTRACT

To detect collisions among objects and other events in the environment is a considerable problem in the simulation area, since the computer resources required grow in a proportional scale to the quality to be achieved. Therefore, when it comes to quality and performance, it is important to pay attention when choosing the right technic to do such tasks. However, it is possible to set up the balance between power and efficiency using quadtrees and octrees. These hierarchical data structures combine both characteristics, performance and quality, so they are able to attend to many needs. Showing itself as a flexible and, at the same time, efficient solution to the spacial events indexing. In order to show the efficiency of these structures, in this term paper will be shown a study around both collision detection techniques, the one using the mentioned data structures and the regular method. The results will be analyzed, compared and discussed.

Keywords: Collision Detection. Quadtree. Octree. Computer Graphics.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

2D	Bidimensional
3D	Tridimensional
API	*Application Programming Interface*
CRT	*Cathode Ray Tube*
GUI	*Graphical User Interface*
IA	Inteligência Artificial
IBM	*International Business Machine Corporation*
UFRJ	Universidade Federal do Rio de Janeiro

SUMÁRIO

1 INTRODUÇÃO

Por meio da Computação Gráfica é possível representar virtualmente o mundo real, utilizando-se modelos matemáticos. A evolução tecnológica possibilita aprimorar estas representações, modelos da Física, da Química, da Biologia e demais ciências. Assim são aplicadas propriedades, leis, modelagens geométricas, controle de movimentos e renderização para que possam modificar o estado e o movimento dos objetos gerados por computador.

O uso da Computação Gráfica para a visualização da variação de comportamento de objetos reais e simulados tem se tornado cada vez mais popular para as ciências e engenharias. Pode-se usar tais recursos para representar modelos matemáticos de fenômenos tais como simulação de fluidos, relatividade, reações químicas e nucleares, sistemas fisiológicos e funções de órgãos, deformação de estruturas mecânicas em reação à vários tipos de pressão, dentre outros (FOLEY, 1995). Sendo assim, a Computação Gráfica tornou-se indispensável no campo da simulação, pesquisa e desenvolvimento das mais diversas áreas.

Neste contexto, a detecção de colisão tem sido um problema fundamental na animação por computadores, modelagem física, modelagem geométrica e robótica. Nestas aplicações, interações entre objetos que se movem são modeladas com limites dinâmicos e análise de contato.

Uma colisão é definida pela interesecção da área ou volume de dois ou mais artefatos situados no espaço, de modo que a movimentação destes objetos em uma situação de simulação deve ser restrita por várias interações, incluindo as colisões.

Uma forma de representação da detecção de colisão entre objetos possibilita a indexação dinâmica do espaço onde estes se encontram, para que a verificação de interação entre eles possa ser feita, de modo que as colisões possam ser computadas. Dependendo da estrutura de dados selecionada para a representação, diferentes modelos de movimentações dos objetos também podem ser empregadas. Porém, todas estas representações têm seu custo computacional.

Há inúmeras estruturas de dados para representações de epaço bidimensional e tridimensional e que podem ser aplicadas à detecção de colisão

entre objetos. Estruturas de dados hierárquicas para representações bidimensionais e tridimensionais possibilitam testes recursivos nas regiões representadas, muitas vezes reduzindo a complexidade computacional envolvida. Soluções para esta indexação podem ser feita por meio de estruturas de dados hierárquicas chamadas *quadtrees*, para representações bidimensionais, e *octrees*, para representações tridimensionais.

Quadtree é uma estrutura de dados utilizada para representações bidimensionais onde qualquer representação pode ser subdividida em quatro quadrantes, sendo que cada quadrante pode ser subdividido novamente em quatro subquadrantes e assim sucessivamente de forma recursiva até que uma condição de parada seja alcançada. Na *quadtree*, a estrutura é representada por um nó denominado pai, enquanto os quatro quadrantes são representados por quatro nós filho (WAGNER, 2012).

Octree é uma estrutura de dados utilizada para representações tridimensionais em forma de árvore na qual, a partir do nó raiz, cada nó pai tem exatamente oito filhos. Cada um destes nós é representado, no espaço, por um bloco em forma de cubo e seus filhos são, nada mais do que, uma divisão interna do próprio bloco. Em outras palavras, o nó raiz é dividido em oito blocos iguais; cada um destes blocos é então subdivido em mais oito nós e assim sucessivamente até que uma condição de parada seja alcançada (SUTER, 2011).

Uma vez formada a árvore em torno de um evento situado no espaço, é possível, a partir da informação contida em cada folha conhecer a área ou o volume ocupado por esta seleção e a sua posição espacial. A partir disso é possível, recursivamente, realizar simulações de colisão entre dois ou mais objetos devidamente indexados no espaço.

1.1 OBJETIVO GERAL

Aplicar estruturas de dados hierárquicas *quadtrees* e *octrees* para realizar detecção de colisão de polígonos em ambientes bidimensionais e tridimensionais.

1.2 OBJETIVOS ESPECÍFICOS

Este trabalho tem como objetivo satisfazer os seguintes itens:

a) estudar o problema de Detecção de Colisão em Ambientes Bidimensionais e Tridimensionais;
b) compreender as estruturas de dados hierárquicas *quadtrees* e *octrees*;
c) demonstrar a construção de *quadtrees* e *octrees* para representação e a indexação de eventos no espaço;
d) estudar métodos eficientes para renderização das estruturas de dados para representações tridimensionais;
e) avaliar e comparar características resultantes das técnicas de construção de *quadtrees* e *octrees* para representação e a indexação de eventos no espaço aplicadas detecção de colisão de polígonos em ambientes bidimensionais e tridimensionais, juntamente com seus algoritmos, baseando-se em métricas objetivas tais como custo computacional;
f) demonstrar os resultados comparativos por meio da implementação dos algoritmos utilizados.

1.3 JUSTIFICATIVA

Com a necessidade de simulações físicas cada vez mais precisas na Computação Gráfica, sejam para a simulação, entretenimento e afins, algoritmos eficientes para determinadas tarefas são cada vez mais exigidos.

Realizar testes de colisão com superfícies e objetos é algo oneroso do ponto de vista computacional. E se tratando de superfícies muito grandes isto se torna inviável (ALMEIDA, 2011).

O propósito deste trabalho é discutir e implementar os métodos de representação de estruturas de dados hierárquicas baseados em *quadtrees* e *octrees*, para realizar eficientemente a detecção de colisão de polígonos em ambientes bidimensionais e tridimensionais de tamanho variável. Estas estruturas foram escolhidas por serem potencialmente promissoras, baseadas em regiões

hierárquicas, de forma que possam ser utilizadas para segmentar as regiões de interesse, com menor esforço computacional, uma vez que já são particularmente flexíveis o bastante para indexar eventos no espaço.

A motivação deste Trabalho de Conclusão de Curso vem da necessidade de simular modelos da Física para movimentos de corpos em duas e três dimensões e seus respectivos momentos de inércia, impulso e energia em colisões elásticas de forma realística.

Quadtrees e *octrees* são uma possível solução para tal problema. Por se tratarem de estruturas de dados em árvore, a criação de *quadtrees* e *octrees* e a manipulação de seus dados pode ser realizada de maneira flexível, uma vez que métodos recursivos podem ser desenvolvidos para tal tarefa.

1.4 ESTRUTURA DO TRABALHO

Este trabalho é divido em três partes: o levantamento bibliográfico, o trabalho proposto e a metodologia utilizada para a solucionar o problema e alcançar os objetivos, e os resultados obtidos.

Na primeira parte é realizado um levantamento bibliográfico de modo a facilitar o entendimento acerca dos conceitos de estruturas de dados, computação gráfica, detecção de colisão e duas estruturas de dados específicas, as *quadtree* e *octree.*

No tópico sobre estruturas se dá ênfase às estruturas hierárquicas e sua terminologia, conceitos de recursão e recursividade aplicada em árvores.

No capítulo seguinte são explorados alguns conceitos de computação gráfica como sua origem e conceitos relacionados aos três maiores campos da área: processamento, síntese e análise. Logo a seguir são descritos o espaço bem como o sistema cartesiano de coordenadas.

O capítulo seguinte refere-se à detecção de colisão, que salienta os conceitos básicos sobre a detecção de colisão e suas técnicas mais fundamentais, utilizando-se de alguns exemplos. Na sequência são abordadas *octrees* e *quadtrees,* bem como sua estrutura e aplicações e como se relacionam com os conceitos previamente citados sobre detecção de colisão.

Na segunda metade do trabalho é definido o sistema proposto e todo o estudo realizado acerca deste, de modo que uma vez mais as estruturas de dados *quadtree e octree* são abordadas, entretanto, de maneira mais aprofundada.

Posteriormente é abordado o algoritmo cuja estrutura serviu de base para o desenvolvimento do ambiente de testes e obtenção dos resultados.

E, por fim, são abordados e comentados os resultados obtidos realizando uma análise comparativa quantitativa acerca destes, que são seguidos das considerações finais.

2 ESTRUTURAS DE DADOS

A organização de dados envolvidos em um problema e o desenvolvimento de algoritmos para operar sobre dados são os dois maiores aspectos do desenvolvimento de software. A redução desta problemática se deu com o surgimento das estruturas de dados, que são um conjunto de técnicas e algoritmos criados para para armazenar e manipular dados, respectivamente (NYHOFF, 2005, tradução nossa).

Uma estrutura de dados é um conjunto de dados que contém uma relação entre si. Estes dados podem ser homogêneos ou heterogêneos e são organizados de modo que possam ser acessados e manipulados de forma eficiente através de rotinas de manipulação. Há diversos tipos de estruturas de dados. Sua arquitetura e a maneira com a qual os dados são dispostos varia de acordo com a sua proposição e as tarefas que deverá desempenhar. Neste ponto, pode-se dizer que há estruturas de dados com finalidades bem específicas (NYHOFF, 2005, tradução nossa).

Como exemplo, pode-se citar o vetor, uma estrutura linear, cujo conceito é um dos mais básicos dentre as estruturas de dados. Sua função é alocar, dinâmica ou estaticamente, uma quantidade de espaço na memória principal para armazenar dados homogêneos (NYHOFF, 2005, tradução nossa).

Estas informações podem ser acessadas diretamente através de índices e de rotinas que trabalhem com estes índices, abstraindo, assim, tal tarefa para um nível mais alto de programação.

2.1 ESTRUTURAS DE DADOS HIERÁRQUICAS

As estruturas de dados hierárquicas, também chamadas de Árvores, como o próprio nome sugere, são feitas à semelhança das árvores encontradas na natureza. Possuem tronco e ramos principais que podem se subdividir em novos ramos, chamados de ramos secundários, e assim por diantes até chegarem nas folhas. Na Ciência da Computação, no entanto, tal abstração das árvores da natureza é desenhada de cima para baixo, deixando, assim, a raiz no topo do diagrama e não na base (KOFFMAN; WOLFGANG, 2008).

Os elementos de uma árvore são organizados de maneira hierárquica de modo a respeitar um determinado critério e ao contrário do que acontece nas estruturas lineares, cada elemento pode ter mais de um sucessor.

Para facilitar o entendimento, pode-se imaginar uma lista encadeada como o modelo mais básico de árvore em que cada nó possui apenas um único filho (PENTON, 2003, tradução nossa).

Entretanto, a complexidade destas estruturas de dados é superior à de uma lista encadeada na qual cada elemento possui apenas um sucessor; de modo que, em uma árvore, cada nó pode possuir um ou mais sucessores, mas apenas um predecessor.

2.1.1 Terminologia

Para descrever os elementos que compõem uma árvore utiliza-se de uma terminologia semelhante à utilizada para detalhar as árvores encontradas na natureza.

Uma vez que, como abordado anteriormente, uma árvore na Ciência da Computação é desenhada de ponta-cabeça, sua raiz fica no topo; o que implica que esta cresce para baixo. A partir da raiz são dispostos os elementos da árvore que são comumente chamados de *nós* e que estão diretamente relacionados aos seus, caso haja mais de um sucessores. O sucessor de um *nó* é chamado de *filho* e o seu antecessor é chamado de *pai*, assim como *nós filhos* do mesmo *pai* são chamados de *irmãos*. As ligações entre *pai* e *filho* são estabelecidas através de ramos (KOFFMAN; WOLFGANG, 2008).

Uma árvore possui duas regiões, a interna e a externa. A região interna é composta pelos elementos que estão situados no interior da árvore e que possuem sucessores; já a região externa da árvore é composta por elementos situados nos limites da estrutura de dados, não possuindo, portanto, sucessores. Um nó sem sucessor é chamado de *folha*.

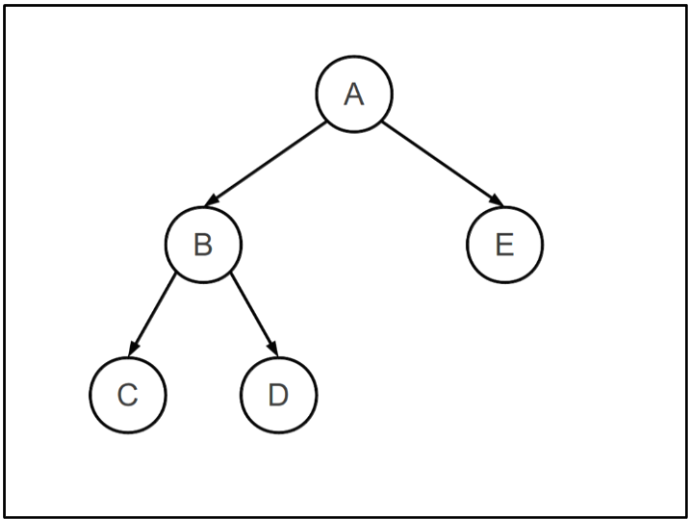
A distância que separa um nó da raiz é chamada de *nível* ou de *profundidade* e é determinada pela quantidade de elementos que o antecedem (KOFFMAN; WOLFGANG, 2008).

Segundo Drozdek (2002), os nós de uma árvore são separados por arcos e deve ser possível alcançar cada nó a partir da raiz por meio de sequência arcos

que deve ser única chamada de *caminho*. O comprimento deste *caminho* é definido pela quantidade de arcos que possui, bem como a *profundidade* de um nó é igual ao comprimento entre a raiz até o nó mais 1.

Na figura 1, por exemplo, é representada uma árvore simples. Deve-se considerar cada círculo como um nó, assim como ramos as linhas que conectam círculos entre si. No caso a seguir, o nó *B*, por exemplo, é um nó interno, de nível 1, é filho de *A*, irmão de *E* e pai de *C* e *D*.

Figura 1 – Exemplo de árvore



Fonte: Do autor.

2.1.2 Natureza recursiva das árvores

Árvores são consideradas estruturas de dados recursivas. Isso se dá porque é possível encontrar outras árvores dentro de suas estruturas. Os nós de uma árvore podem ser interpretados cada um como uma raiz de uma nova árvore. (PENTON, 2003, tradução nossa).

Na figura 1, o nó *A* é a raiz de toda a árvore. Entretanto, dada uma análise mais minuciosa, pode-se notar que o nó *B,* por exemplo, é raiz de uma árvore menor, cujos filhos são *C* e *D.*

Dada esta constatação, é possível usar técnicas que compreendem recursividade, que será abordada a seguir, para poder trabalhar com estas estruturas.

2.2 RECURSIVIDADE

Recursão é a habilidade de uma função chamar a si mesma. Embora seja uma definição simples, é um conceito complexo para algumas pessoas entenderem pois sua complexidade reside em sua aplicação e não em sua compreensão (PENTON, 2003, tradução nossa).

Segundo Drozdek (2002), a definição de novos objetos e conceitos deve acontecer seguindo uma regra básica de que sua definição deve seguir conceitos ou definições que já tenham sido estabelecidos ou que sejam óbvios. Se um objeto for definido de acordo com regras que ele mesmo define, é criada, então, uma contradição lógica, o que leva a círculo vicioso. Entretanto, há conceitos computacionais que se autodefinem e para que sejam possíveis, são estabelecidas restrições formais às definições em si, possibilitando que estes não quebrem a regra.

Na computação, uma função recursiva é caracterizada por chamar ela própria, seja direta ou indiretamente. Embora isso possa parecer um problema, é possível que esta trabalhe apropriadamente de modo a retornar um resultado correto. A recursão em uma função é realizada com o uso de uma pilha que trabalha em tempo de execução, cujo responsável pelo seu gerenciamento é o sistema operacional. A idéia de se usar pilha para se trabalhar com recursão foi sugerida Edsger Djisktra (DROZDEK, 2002).

2.2.1 Recursividade em árvores e a abordagem top-down

Há diversas estratégias relacionadas à resolução de problemas. No caso das árvores, a problemática se trata da melhor maneira de percorrê-la, acessar seus dados e manipulá-los eficientemente.

Uma solução viável para se trabalhar com árvores é a abordagem *top-down,* também chamada de *dividir e conquistar,* que consiste em acessar a árvore partindo do nó raiz em direção aos nós folhas. O termo *dividir e conquistar* remete ao processo de se dividir um problema em problemas menores e resolvê-los de maneira independente. Uma vez que os pequenos problemas sejam resolvidos, a solução destes é combinada de modo a chegar na resolução do problema mor (PREISS, 1997).

Em uma árvore, a árvore em si é o problema e as suas subárvores possíveis são os pequenos problemas. É neste ponto em que entra a abordagem *top-down,* capaz de resolver cada um destes pequenos problemas independentemente.

Segundo Drozdek (2002), no caso de uma busca por um determinado valor *N* em uma árvore binária, na qual cada nó possui no máximo dois filhos, os filhos com valores menores do que seu pai se posicionam na esquerda e os filhos com valores maiores do que seu pai se posicionam na direita, um algoritmo recursivo – que também é um *top-down* - executaria os seguintes passos:

a) verifica se o valor do nó apontado atualmente é igual ao valor *N* a ser localizado;
b) se *N* for menor do que o valor do nó apontado pelo algoritmo, então o vá para o filho esquerdo do nó apontado e executa novamente o passo *a*;
c) se *N* for maior do que o valor do nó apontado pelo algoritmo, então vá para o filho direito do nó apontado e executa novamente o passo *a*;
d) se *N* for igual ao valor do nó apontado, a busca termina a retorna a informação de que o valor foi encontrado;
e) se não for possível continuar a busca, o que indica que não há nó com um valor correspondente para *N,* o algoritmo retorna a informação de que o valor não foi encontrado.

3 COMPUTAÇÃO GRÁFICA

Computação gráfica é vista como uma das ramificações da Ciência da Computação, cujo objetivo é lidar com a teoria e tecnologia para a síntese computacional de imagens (XIANG; PLASTOCK, 2000, tradução nossa).

A computação gráfica só se popularizou a partir da década de 1980. Até então o alto custo de *hardware* aliado à pequena quantidade de *software* de fácil uso eram um fator determinante que reduziu o campo de atuação da área, tornando-a muito pequena e especializada. A comercialização de computadores com placas gráficas acopladas - como os Apple Macintosh e os computadores da *International Business Machine Corporacion* (IBM) - popularizou o uso de gráficos *bitmap*, ou mapa de bits, para a interação usuário - computador. Uma vez que essa tecnologia se torno popular e barata, houve a criação de *software* fáceis de usar, assim como aplicações com interface gráfica e de baixo custo. Estas aplicações com interface gráfica de usuário - *Graphical User Interface* (GUI) - possibilitaram que milhões de novos usuários pudessem ter acesso a aplicações de baixo custo e simples de serem usadas, como editores de texto e programas para desenho, por exemplo (FOLEY, 1995, tradução nossa).

Antes de ir para os computadores, era usada para demonstrar dados em cartões perfurados e em telas de tubo de raio catodo - *Cathode Ray Tube* (CRT). Seus recursos são usados em uma quantidade crescente de áreas, como Matemática, Física, Arquitetura, Engenharia, entre outras e sua função engloba a criação, armazenamento e manipulação modelos e imagens de objetos, abstraídos ou não da realidade. Atualmente a computação gráfica é vista como uma ferramenta interativa, na qual é possível controlar seus gráficos por meio de dispositivos de entrada como teclados, mouse ou telas sensíveis ao toque, por exemplo (FOLEY, 1995, tradução nossa).

Para se abordar a computação gráfica é necessário que se aborde o que é imagem, em especial a imagem digital.

Azevedo e Conci (2003, p. 3) dizem que “a imagem digital é uma representação de uma imagem em uma região discreta, limitada através de um conjunto finito de valores inteiros que representam cada um dos seus pontos.”

Conforme Azevedo e Conci (2003), essas imagens podem ser caracterizadas de acordo com sua representação no espaço, sendo unidimensionais, bidimensionais ou tridimensionais; o conteúdo de seus pontos, sendo binárias, monocromáticas, multibandas ou coloridas; e a sua forma de descrição, sendo vetoriais ou matriciais.

3.1 ÁREAS DA COMPUTAÇÃO GRÁFICA

A computação gráfica é dividida em três importantes áreas de estudo: processamento de imagens, análise de imagens, síntese de imagens. Estas áreas serão abordadas a seguir.

Há também uma área menor, que pode ser considerada uma sub-área da análise de imagens, e será abordada juntamente com as demais, é chamada de visão computacional.

3.1.1 Processamento de imagens

O processamento de imagens está relacionado à manipulação de imagens uma vez que tenham sido geradas, seja através de captura ou síntese. Neste processo, os dados de uma imagem são fornecidos como entrada, processados e, na sua saída, são usados para criar uma imagem resultante.

Os processos que manipulam dados de imagens como variáveis de entrada, resultando em uma nova imagem como saída, pertencem à área de processamento de imagem. Esta área inclui tópicos como aplicação de filtros, tratamento de ruídos, restauração de imagens, entre outros e é frequentemente utilizada para extrair informações das imagens, o que pode auxiliar no processo de reconhecimento de padrões, extração do conhecimento das imagens ou visão computacional (AZEVEDO; CONCI, 2003).

3.1.2 Análise de Imagens

Como visto, o processamento recebe uma imagem como entrada e produz uma imagem como saída. A análise de imagem funciona de maneira

semelhante, porém, o que a torna diferente é que, ao invés de produzir uma imagem, ela produzir outros tipos de saída.

O resultado desta análise é um conjunto de informações úteis que descrevem, de maneira geralmente numérica, os pontos importantes da imagem analisada. A extração dessas informações é realizada por um algoritmo que possui teorias e métodos responsáveis pelo processo analítico. Este processo respeita critérios pré-definidos (AZEVEDO; CONCI, 2003).

3.1.3 Síntese de Imagens

À área de síntese de imagens competem as teorias e práticas referentes a geração de imagens.

Os algoritmos responsáveis por tal tarefa recebem informações como entrada e geram imagens como saída. Pode-se dizer que esta área remete ao processo inverso da análise de imagens, que recebe imagens como entrada e produz informações como saída.

A síntese de imagem é usada em aplicações com o objetivo de gerar imagens com base em primitivas geométricas como linhas, curvas e superfícies de modo a representar objetos do mundo real. Na técnica de síntese, o computador transforma em imagem informações que de outra maneira seria possível interpretar; pode-se citar como exemplo imagens termográficas, de ressonância magnética, de ultrassons etc (AZEVEDO; CONCI, 2003).

3.1.4 Visão computacional

A visão computacional pode ser considerada uma sub-área da análise de imagem e é responsável pelas técnicas que envolvem o sistema de visão do computador, ou seja, sua capacidade de perceber visualmente o que está ao seu redor.

Segundo Azevedo e Conci (2003), a área de visão computacional é responsável por extrair informações, identificar e classificar objetos presentes nas imagens. Esses sistemas de visão computacional são usados em conjunto com técnicas de Inteligência Artificial (IA) para realizar tarefas em diversas áreas como,

por exemplo, reconhecimento facial, inspeção em linhas de montagem ou auxiliar a movimentação de robôs.

Sabe-se que o olho humano é o órgão responsável pelo sentido da visão e que, através da captura de energia luminosa, consegue estabelecer relações temporais e espaciais entre os objetos que estejam em sua linha de visão. Essa energia luminosa ao passar pelo olho é convertida para sinais elétricos que são enviados para o cérebro através do nervo óptico, que possui cerca de um milhão de fibras nervosas. A retina é a parte mais importante o processo de percepção visual pois possui mais de 100 milhões de sensores responsáveis pela captura de estímulos luminosos (AZEVEDO; CONCI, 2003).

O sistema visual do ser humano não é apenas uma lente com a função passar para o cérebro tudo o que vê. Ambas as tarefas, tanto a de percepção quanto a de processamento do que é visto estão intimamente relacionadas ao passo que, para o homem, é muito difícil desassociar um processo do outro. Isto se dá devido ao fato de que o homem interpreta as imagens e as reconhece porque aprendeu a reconhecê-las e interpretá-las assim (AZEVEDO; CONCI, 2003).

Na visão computacional, assim como na visão humana, há a necessidade de outros processos que não só os de captura de imagem de modo que uma imagem possa ser interpretada.

O ser humano gera relações entre os objetos que vê de maneira cognitiva pois a percepção e o processamento do que sua visão capta estão diretamente relacionados, não só entre si, mas com as experiências pelas quais já passou previamente.

O computador, por outro lado, naturalmente não tem a capacidade de interpretar o que lhe é mostrado. Um sistema computacional depende de outras informações de entrada além da imagem a ser processada ou analisada. Muitas destas informações, embora muitas vezes passem desapercebidas para o ser humano, são interpretadas pelo computador através de valores numéricos.

3.2 O SISTEMA CARTESIANO DE COORDENADAS

Para se entender com clareza como um sistema computacional lida com objetos geométricos é necessário compreender a interação entre os números e a

geometria. Considerando que o ser humano possui uma intuição geométrica que possibilita entender descrições verbais de objetos como *linha* e *ângulo*, também há a capacidade computacional de se manipular números. O problema é como expressar essas idéias geométricas de forma numérica de modo que o computador possa interpretá-las (XIANG; PLASTOCK, 2000. tradução nossa).

O computador, como se sabe, trabalha de forma numérica e para que seja possível o entendimento de uma idéia geométrica é necessária a tradução desta idéia para uma ou um conjunto de expressões numéricas. Essa tradução é possível através de um sistema de coordernadas.

O Sistema Cartesiano de Coordenadas é apenas uma das muitas maneiras de representar numericamente formas no espaço e será abordado a seguir, especificamente nos espaços bidimensional (2D) e tridimensional (3D).

3.2.1 Sistema Cartesiano Bidimensional

O espaço bidimensional é, como o próprio nome sugere, um espaço com duas dimensões, plano e tudo que nele é representado é medido através de duas grandezas.

De acordo com o Sistema de Cartesiano de Coordenadas, um plano 2D pode ser representado graficamente por duas linhas perpendiculares que se cruzam em um ponto chamado de *ponto de origem*. Estas linhas são tradicionalmente chamadas de eixo *x* e eixo *y* (XIANG; PLASTOCK, 2000, tradução nossa).

Perpendiculares ao eixo *y* e ao eixo *x* há infinitas linhas dispostas de modo a formar uma grade sobre o plano. Qualquer ponto *P* no plano está situado em uma intersecção formada por uma linha perpendicular ao eixo *x* e uma linha perpendicular ao eixo *y*. Os valores de *x* e *y* associados a qualquer ponto *P* posicionado no plano definem sua posição no Sistema Cartesiano 2D e são representadas em formas de pares (*x*, *y*) (XIANG; PLASTOCK, 2000, tradução nossa).

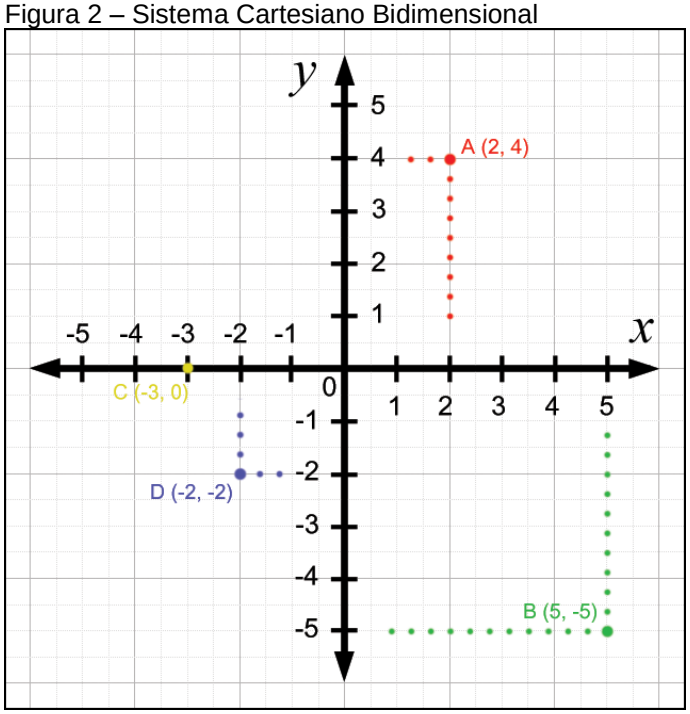
Com a utilização destes pontos é possível desenhar qualquer objeto bidimensional no plano. Esta representação numérica possibilita cálculos como os de perímeto e área e também a aplicação de transformações bidimensionais como escalamento, rotação e transladação.

Segundo Xiang e Plastock (2000, tradução nossa), no Plano Cartesiano, a distância entre quaisquer dois pontos P_1 e P_2 com as respectivas coordenadas (x_1, y_1) e (x_2, y_2) pode ser medida com o uso de uma reta, cujo comprimento pode ser definido através da fórmula:

$$D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

A Figura 2 exemplifica um Sistema Cartesiano Bidimensional com pontos *A*, *B*, *C* e *D* e suas respectivas coordenadas.

Figura 2 – Sistema Cartesiano Bidimensional



Fonte: Do autor.

3.2.2 Sistema Cartesiano Tridimensional

O espaço tridimensional é composto por três dimensões e tudo que nele é representado possui três grandezas, comumente chamadas de *largura, altura* e *profundidade.*

Graficamente, este espaço pode ser representado através do sistema Cartesiano Tridimensional e é composto de três linhas, mutuamente perpendiculares e que cruzam o *ponto de origem*, assim como no Espaço Cartesiano 2D, com a diferença de que há um eixo a mais. Estes eixos são definidos como eixo *x,* eixo *y* e

eixo *z* e seus nomes são dispostos nos finais positivos de cada eixo (XIANG; PLASTOCK, 2000, tradução nossa).

Segundo Xiang e Plastock (2000, tradução nossa), qualquer ponto *P* dentro do espaço 3D é representado através de triplas (*x*, *y*, *z*) que representam as seguintes propriedades:

a) o eixo *x* representa a distância que o ponto *P* acima ou abaixo em relaçao ao eixo *yz;*
b) o eixo *y* representa a distância que o ponto *P* acima ou abaixo em relaçao ao eixo *xz;*
c) o eixo *z* representa a distância que o ponto *P* acima ou abaixo em relaçao ao eixo *xy.*

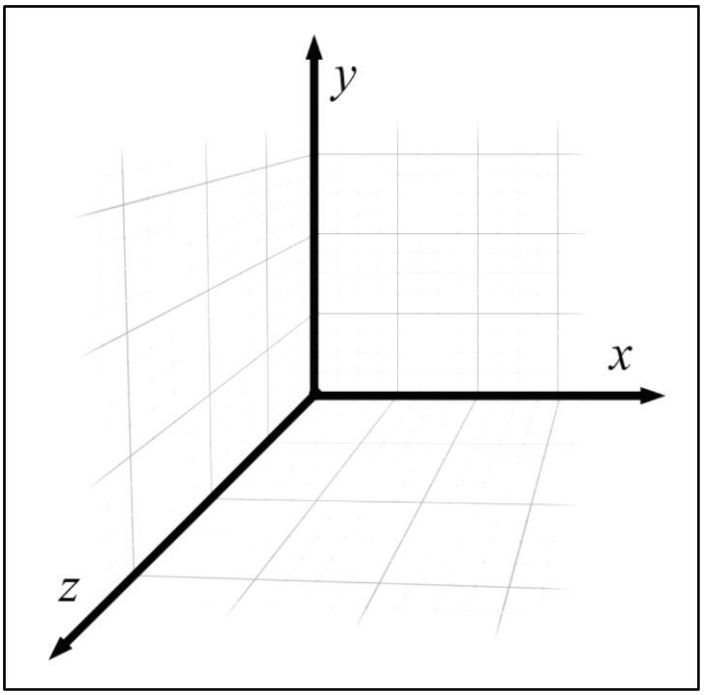
A existência de mais um eixo possibilita a aplicação de fórmulas para outros cálculos como o de volume, por exemplo.

A adição de um eixo não muda o fato de que a menor distância entre dois pontos é uma reta, portanto para se calcular a distância entre quaisquer dois pontos P_0 e P_1 situados no espaço com as respectivas coordenadas (x_0, y_0, z_0) e (x_1, y_1, z_1) , usa-se a fórmula:

$$D = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2}$$

A figura 3 representa o sistema Cartesiano 3D. Considerando que não há volume negativo, geralmente há a omissão dos quadrantes negativos de cada eixo, todo objeto posicionado no espaço surgirá a partir da origem rumo ao espaço positivo. O posicionamento dos eixos é variável, em especial o eixo *z* que pode, ao invés de apontar para fora da tela, em direção ao espectador, apontar para dentro da tela, na direção para qual o espectador olha.

Figura 3 – Sistema Cartesiano Tridimensional



Fonte: Do autor.

4 DETECÇÃO DE COLISÃO

Uma colisão acontece no momento em que há a intersecção da área ou volume de dois ou mais objetos situados no espaço. A detecção de tal intersecção acontece através de cálculos que, em meio computacional, assume que os artefatos situados no espaço possuam a propriedade de sólidos.

Para isso é necessário cumprir duas etapas (MARROQUIM, 2011):

a) detecção da colisão: para poder precisar quando e onde a colisão acontece;
b) resolução da colisão: para poder tratar a colisão e estabelecer onde os objetos estarão posicionados depois da colisão.

Segundo Marroquim (2011), os problemas frequentemente encontrados nos processos de detecção de colisão estão relacionados a velocidade dos objetos no espaço, a complexidade geométrica destes objetos, técnicas para detecção de colisão em sistemas com prioridade de tempo real ou ao alto custo computacional necessário para que a detecção seja satisfatoriamente realizada. Neste último caso, tem-se como exemplo o caso de se haver vários objetos no espaço e ser necessária a realização de testes entre todos os objetos; a complexidade de tal tarefa se dá conforme a seguinte notação matemática:

$$n \times (n-1) \rightarrow O(n^2)$$

Marroquim (2011) diz que as duas técnicas mais básicas usadas para detecção de colisão são o *teste de sopreposição* e o *teste de intersecção*. A primeira verifica a colisão a *posteriori*, ou seja, depois, verificando se esta já aconteceu; a segunda verifica a *priori*, ou seja, antes, de modo a prever se a colisão ocorrerá.

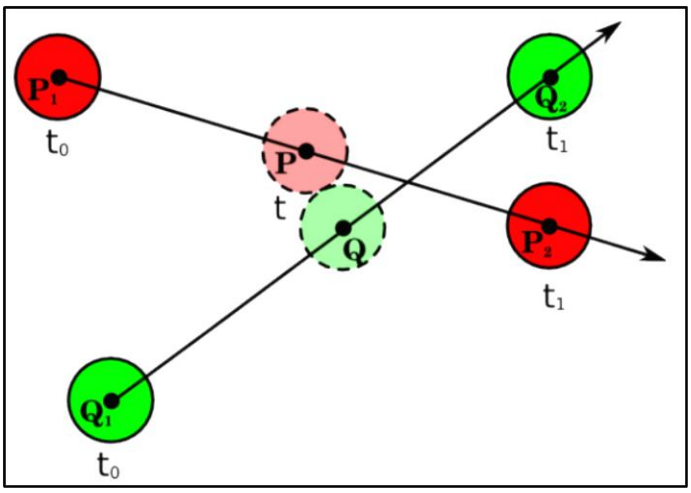
4.1 TESTE DE INTERSECÇÃO

Em um teste espacial realizada a *priori* é possível detectar uma colisão entre objetos no espaço antes que esta efetivamente ocorra. Este teste é realizado através do posicionamento de uma área ou volume de varredura em torno dos objetos situados no espaço. A detecção de colisão acontece por meio da varredura

do espaço a ser percorrido pelo objeto através da utilização de métodos geométricos responsáveis por determinar o estado inicial, intermediário e final do objeto, buscando possíveis interesecções desta área ou volume ou outros objetos (MARROQUIM, 2011).

O teste de intersecção é mais robusto em relação ao teste de sobreposição e não impõe limites relacionados à velocidade do movimento dos objetos, tamanho ou tempo. Os problemas frequentemente encontrados estão relacionados à detecção de colisão em casos não triviais e a técnica assume velocidade constante dos objetos no espaço, ou seja, não há aceleração (MARROQUIM, 2011).

Figura 4 – Intersecção de dois objetos no plano



Fonte: MARROQUIM (2011)

Na Figura 4 é mostrado um teste de intersecção que verifica a posição inicial e final de P_1 e P_2 e estima o ponto que haverá colisão entre os dois objetos.

Segundo Marroquim (2011), embora o teste de intersecção seja mais eficiente que o de sobreposição, ambos sofrem dos mesmos problemas: enfrentam dificuldades ao realizar testes com geometrias demasiado complexas e executam todas as combinações de testes possíveis, o que acarreta em um custo computacional elevado.

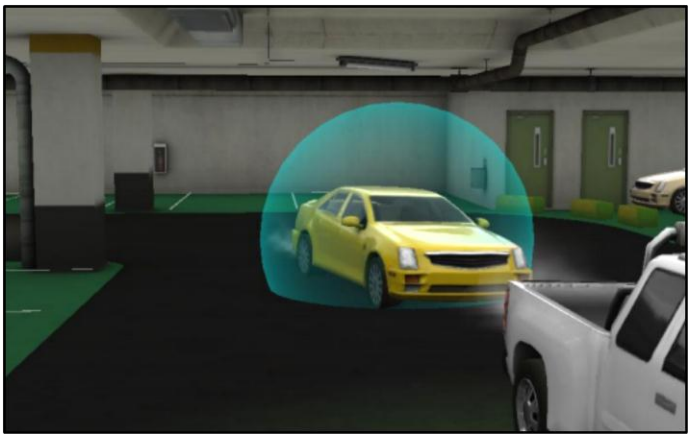
4.1.1 Volumes envolventes e a divisão espacial

Como uma variação da técnica de intersecção, frente aos problemas nela encontrados, é possível a aplicação de volumes envolventes sobre um objeto no espaço. Trata-se de um volume que encapsula totalmente o objeto e, ao invés de calcular sua tragetória, simplesmente o acompanha de acordo com sua transladação espacial. A técnica assume que se nenhum objeto colide com o volume então não há colisão (MARROQUIM, 2011).

As formas mais utilizadas para envolver objetos no espaço são a esfera, fácil de se testar e possui uma estrutura simples composta por apenas um centro e um raio, e as caixas cuja orientação no espaço pode estar de acordo com os eixos ou com o objeto e podem ser subdivididas de modo a gerar um nível desejado de precisão relacionada as partes de um objetos de estrutura complexa (MARROQUIM, 2011).

A figura 5 ilustra um carro envolvido por uma esfera de colisão.

Figura 5 – Esfera de colisão



Fonte: MARROQUIM (2011)

Com a utilização de caixas de colisão é possível a subdivisão do espaço ocupado por um objeto, propiciando uma maior aproximação do volume real que este ocupa no espaço. Quanto maior a precisão, maior será a eficácia ao se tentar localizar objetos vizinhos. Como esta a representação computacional destas caixas de colisão se dá por meio de árvores, de diversos tipos e características, é possível estabelecer a complexididade computacional de cada técnica de acordo com a árvore utilizada.

Na Figura 6, a seguir, é demonstrado o uso de caixas de colisão. Há dois esqueletos na imagem e o espaço ocupado por seus corpos foi subdivido em diversas caixas de colisão, de modo a possibilitar uma maior precisão na determinação do espaço ocupado por eles. Ademais, na parte inferior da imagem é possível constatar uma colisão entre o membro inferior direito do esqueleto mais à esquerda membro inferior esquerdo do esqueleto mais à direita.

Figura 6 – Caixas de colisão

Fonte: MARROQUIM (2011)

4.1.2 Quadtrees

Black (2011, tradução nossa) define *quadtrees* como um tipo de árvore na qual cada nó é divido em todas as *d* dimensões existentes de modo a resultar em 2^d filhos, ou seja, 4 filhos por nó. O nome *quadtree* surgiu dado o fato de que estas estruturas de dados hierárquicas originalmente foram projetadas para lidar com dados bidimensionais, como imagens, portanto, cada nó possuía exatamente quatro filhos.

Segundo Mendes (2011), *quadtrees* são utilizadas como uma técnica para se representar o espaço de modo a possibilitar a reprodução de imagens 2D. Para se obter uma *quadtree* é necessária a delimitação de uma área no espaço bidimensional e então subdividí-la sucessivamente de modo a formar quadrantes.

4.1.2.1 Representação gráfica de uma *quadtree*

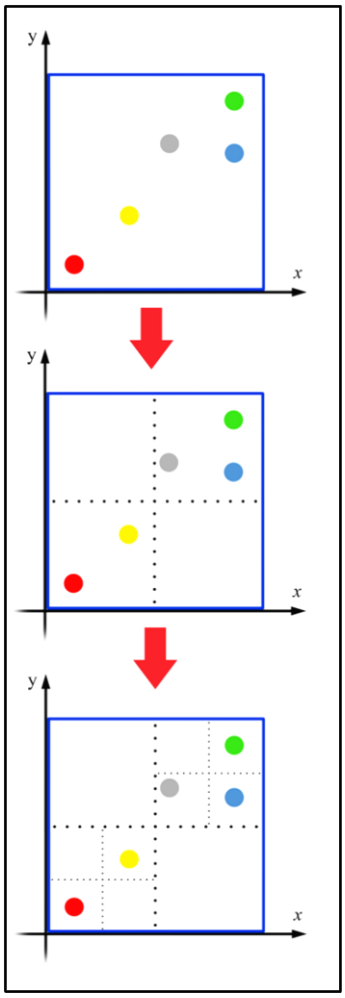
Quadtrees podem ser usadas para diversas funções como indexar a área de um objeto no plano ou armazenar as cores que cada pixel possui em uma imagem. Independente da tarefa, sua estrutura é a mesma. Seus nós podem carregar diferentes tipos de dados e a precisão de uma *quadtree* é determinado pela quantidade de níveis que a árvore possui.

Segundo Suter (1999, tradução nossa), se um nó não é importante então seus filhos também não são.

Quando uma *quadtree* é utilizada para representar uma área no plano, por exemplo, cada quadrante pode assumir apenas três estados: *preenchido, parcialmente preenchido* ou *vazio.* O processo de subdivisão dos quadrantes é realizado de maneira recursiva até que não haja mais quadrantes com o estado *parcialmente preenchido*, restando apenas os de estado *completo* ou *vazio,* ou até que o nível de profundidade desejado alcançado (MENDES, 2011).

A figura 7 exemplifica o processo de subdivisão espacial realizado por uma *quadtree*, ao ponto em que haja apenas um único objeto por quadrante.

Figura 7 - Processo de subdivisão de uma *quadtree*



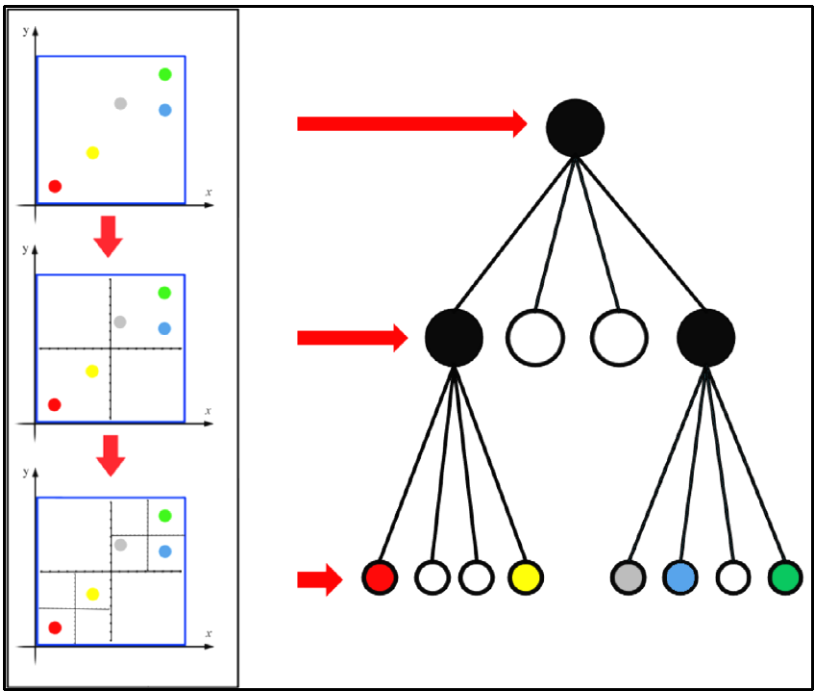
Fonte: Do autor.

4.1.2.2 Detecção de colisão com *quadtrees*

As *quadtree* podem ser usadas para realizar detecção de colisão por meio no método da intersecção de modo que funcionem como caixas de colisão, envolvendo os objetos no espaço.

Para que seja possível a realização de detecção de colisão com *quadtree* é necessário realizar primeiramente a indexação espacial da área na qual se deseja detectar colisões. A indexação espacial ocorre no momento em que a estrutura de dados é criada e, durante este processo, a estrutura recursivamente verifica há a presença de elementos no espaço em que está situada e, recursivamente, os separa entre seus nós filhos.

Figura 8 – indexação espacial com *quadtree*



Fonte: Do autor.

A figura 8, por exemplo, representa um espaço populado por 5 objetos e o processo de indexação espacial em uma *quadtree*. O nó raiz verifica se possui objetos em seus limites, então se subdivide em 4 nós filhos e transfere seus elementos para seus filhos, e o mesmo processo acontece para cada um dos nós filhos: verifica se possui objetos em seus limites para então se subdividir novamente e transferir seus elementos. Este processo ocorre, no exemplo acima, recursivamente até que haja apenas um objeto por nó, de modo que cada nó folha esteja habitado por apenas um objeto.

A indexação do espaço ocorre de modo simples, pois cada nó armazena uma informação ocupacional, informando se o nó está ou não sendo ocupado pelo objeto. Com isto é possível realizar iterações de modo a constatar se há ou se haverá intersecção entre dois nós ocupados por objetos no espaço e com isso, iniciar o tratamento da colisão.

4.1.3 Octrees

Se uma *quadtree* representa uma área pelo fato de possuir duas dimensões, então uma *octree* representa volume, pois possui três. Entretanto não se limita apenas à isto, uma *octree* possui muitas outras funcionalidades como

representar relações espaciais entre objetos, por exemplo. Seus nós podem carregar muitos tipos de informação, entretanto, como é frequentemente usada para indexação de volume em ambientes tridimensionais, os valores mais comumente encontrados em seus nós são valores booleanos que remetem ao estado de o nó estar ou não estar ocupado por um objeto situado no espaço (EDER, 2011, tradução nossa).

4.1.3.1 Representação gráfica de uma *octree*

Em diversos aspectos uma *octree* é semelhante a uma *quadtree*: ela delimita o espaço de interesse e o subdivide até que todos os seus nós estejam homogêneos ou que determinada condição de parada, geralmente profundidade, seja alcançada.

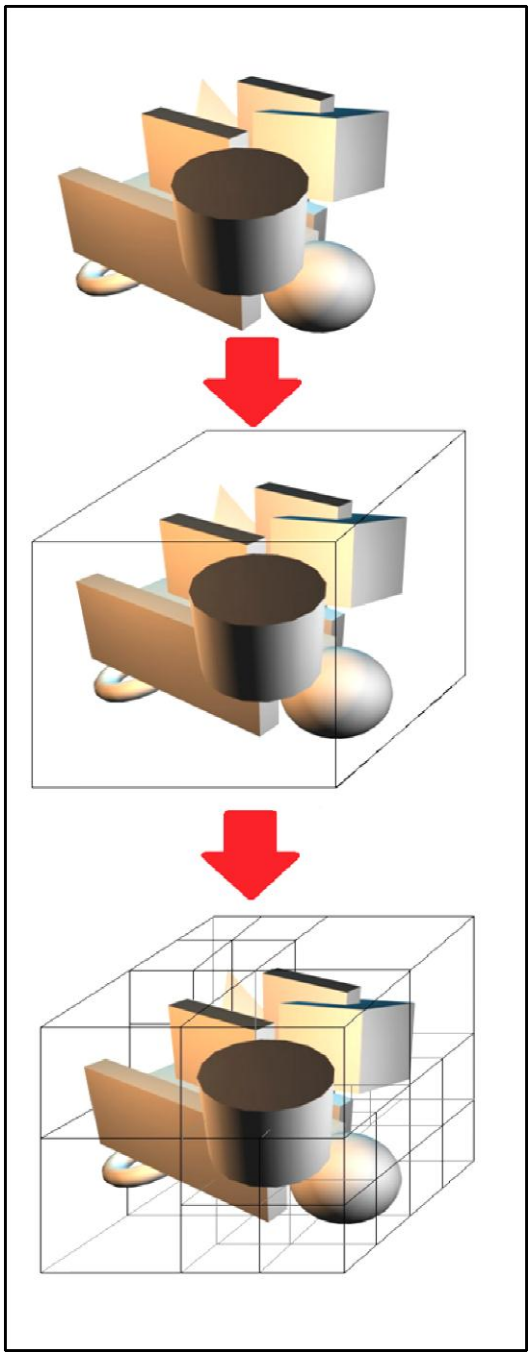
Entretanto, uma vez que a adição de mais uma dimensão torna o processo tridimensional e não mais bidimensional, há um aumento de complexidade na estrutura da árvore.

Black (2011, tradução nossa) definiu anteriormente *quadtrees* como árvores nas quais cada nó possui 2^d filhos sendo *d* a quantidade de dimensões existentes no espaço. Considerando que o espaço é representado por uma variável, esta definição também é válida para se calcular a quantidade de filhos por nó em uma *octree. Octrees* são tridimensionais, ou seja, três dimensões, então *d* assume o valor 3. Portanto a quantidade de filhos por nó em uma *octree* é igual 2^3, o que resulta em 8.

Cada nó de uma *octree* é a representação de um cubo. Quando subdividido, cada cubo é cortado de modo a formar oito cubos menores e assim sucessivamente. Esta divisão do espaço em setores é muito usado para resolver problemas a regiões em um espaço (ALMEIDA, 2011).

A Figura 9 mostra o processo de subdivisão de um espaço ocupado por objetos 3D.

Figura 9 – Processo de subdivisão de uma *octree*



Fonte: SUTER (2011).

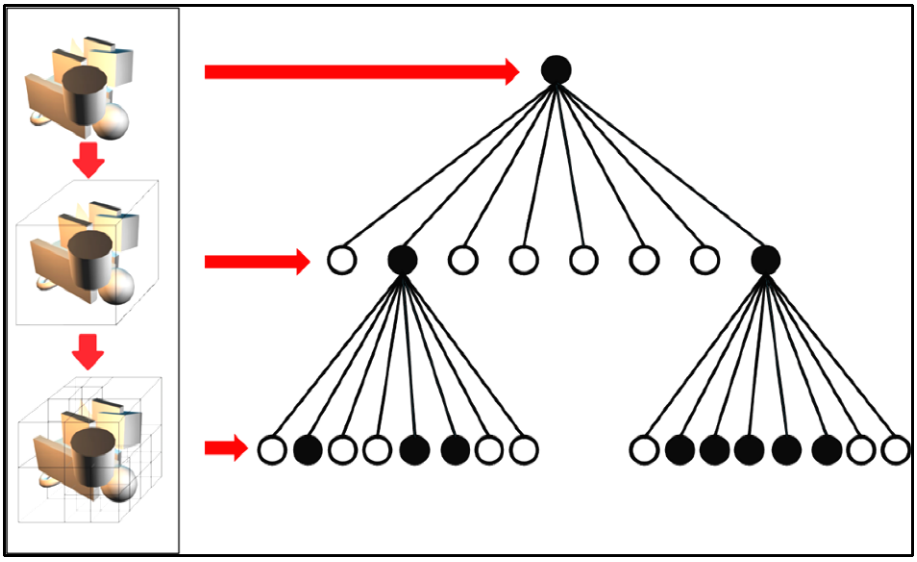
4.1.3.2 Detecção de colisão com *octrees*

O processo de detecção de colisão com *octrees* ocorre de maneira semelhante às *quadtrees* com a diferença de que, como citado anteriormente, há um eixo adicional.

Segundo Almeida (2011), detectar colisão em um espaço 3D sem o uso de *octrees* exigiria executar testes entre todos os elementos do espaço e esta é uma tarefa em $F(n)$, onde *n* seria a quantidade objetos no espaço.

No momento em que uma *octree* é criada para indexar determinado espaço, à medida que seus filhos e os filhos dos filhos são criados, os elementos associados à estes são hierarquicamente transferidos, como na figura 10.

Figura 10 - indexação espacial com *octree*



Fonte: Do autor.

Por exemplo, existe uma *octree* indexando o volume ocupado por um objeto no espaço e a função desta árvore é detectar se há colisão entre este objeto e algum outro. Em média isso exigirá um número de testes igual ao nível de profundidade da estrutura. Considerando que há *n* elementos e a profundidade da *octree* é definida por $\log n$ a realização dos testes terá complexidade equivalente a $F(\log(n))$. Primeiramente serão executados pequenos testes para checar o estado das folhas de modo a identificar com maior precisão onde a colisão ocorre para então ser possível tratá-la.

5 TRABALHOS RELACIONADOS

Há diversas publicações importantes relacionadas ao uso de *quadtree e octree* utilizadas para o processamento de imagens.

Raj Kumar Dash realizou um estudo, publicado em 1993 pela *Dr. Dobb's Journal,* chamado *Image Processing Using Quadtrees: An efficient method for the compression and manipulation of raster images* no qual disserta sobre a utilização de *quadtree* como uma técnica eficiente para a realização de compressão de imagens. Nesta publicação são abordadas *quadtrees,* bem como sua construção, complexidade, lógicas e rotinas de manipulação, diferença entre técnicas utilizadas para imagens monocromáticas e heterocromáticas, utilização de múltiplas *quadtrees*. O código-fonte relacionado é disponibilizado site da *Dr. Dobb's Journal.*

James Foley, professor doutor respeitado internacionalmente na área de computação gráfica. Possui diversos trabalhos publicados na comunidade científica, incluindo livros e artigos, que são considerados clássicos na área.

Ricardo G. Marroquim, professor doutor, trabalha no Laboratório de Computação Gráfica da Universidade Federal do Rio de Janeiro (UFRJ). Possui material disponível na *homepage* da UFRJ relacionado à computação gráfica e a desenvolvimento de jogos eletrônios, o que inclui estudos relacionado à detecção de colisão, diferentes técnicas e sua evolução.

6 AMBIENTE DE DETECÇÃO DE COLISÃO

A proposta deste trabalho é realizar um comparativo de desempenho relacionado à detecção de colisão em ambientes bidimensionais e tridimensionais entre estruturas de dados hierárquicas e algoritmos convencionais. Testes serão utilizados para comparar a eficiencia computacional de ambas as técnicas em diferentes situações.

6.1 METODOLOGIA

Inicialmente foi elaborado um estudo relacionado à arquitetura computacional das estruturas de dados *quadtree* e *octree.* Este estudo será abordado nas seções a seguir, onde serão descritas com mais detalhes as características de ambas as estruturas.

6.1.1 A estrutura de uma *quadtree*

Como já foi abordado nas seções anteriores, uma *quadtree* é uma árvore, ou seja, uma estrutura de dados hierárquica, e pode ser utilizada para, dentre muitas outras finalidades, indexar os elementos no espaço bidimensional. Tal indexação ocorre de maneira recursiva, cujo limite de iterações é arbitrariamente definido.

Computacionalmente, todos os elementos de uma *quadtree*, sejam folhas ou não, são implementados de uma única maneira: uma classe que, a partir de agora, será tratada como nó.

Cada nó possui atributos relacionados à sua posição no espaço, nível atual na árvore em relação ao nó raiz, se possui ou não filhos, quantidade e tipo de objetos que possui e endereço dos filhos, também do tipo nó, bem como métodos relacionados a criação e remoção de filhos e inserção e remoção de objetos destes filhos.

Os valores relacionados ao posicionamento no espaço são valores numéricos com ponto flutuante e delimitam cada nó de uma *quadtree* em limite mínimo estabelecido por um ponto *A (x,y)* e um limite máximo estabelecido por um ponto *B (x,y)* onde *x* e *y* representam, respectivamente a altura e largura, de modo

que todo nó de uma *quadtree* possua a forma de um quadrado. Dados estes dois limites é calculado um ponto central *C (x,y).*

Estes pontos, *A* e *B*, representam o canto inferior esquerdo e o canto superior direito da *quadtree*, já o ponto *C* é utilizado como referência para a subdivisão do nó em nós filhos, caso haja a necessidade.

6.1.1.1 A criação de uma *quadtree*

Uma *quadtree,* no contexto deste trabalho, tem a função de indexar elementos do espaço e organizá-los de modo a facilitar a sua localização e gerenciamento.

Considerando que qualquer objeto situado no espaço possua coordenadas que o localizam, um nó precisa possuir coordenadas, de modo a delimitar sua área de ação. Esta área é utilizada para, além de situar o nó, testar quais objetos fazem parte deste.

No momento em que um objeto do tipo nó é instanciado, é chamado um construtor que recebe como argumento um ponto mínimo, máximo e um valor inteiro correspondente ao nível do nó em relação ao nó raiz; no caso da própria raiz este valor se iguala à zero.

Este construtor inicializa as variáveis do novo nó, fazendo com que seu limite mínimo receba *A* e seu limite máximo receba *B.* Ambos os limites, mínimo e máximo, são utilizados para determinar o ponto central *C*.

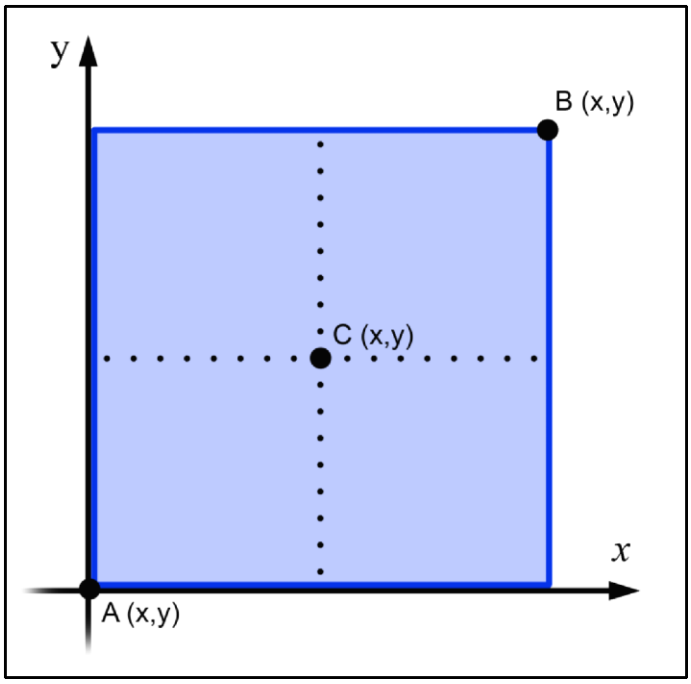
Os valores *x* e *y* de *C* são definidos de acordo com as equações (1) e (2) abaixo, respectivamente:

$$x_C = \frac{(x_A + x_B)}{2} \qquad (1)$$

$$y_C = \frac{(y_A + y_B)}{2} \qquad (2)$$

Na figura 11, a área de um nó é representada pela cor azul e os pontos *A* e *B* são os limites mínimo e máximo, respectivamente. O ponto *C* representa o centro do nó e as linhas pontilhadas separam o nó em quatro quadrantes imaginários. Esses quadrantes, no momento da subdivisão do nó, se tornarão nós filhos.

Figura 11 – Limites de uma *quadtree*



Fonte: Do autor.

Ainda na execução do construtor, a variável binária que define se o nó possui ou não filhos é inicializada. Por padrão, no momento de criação de um nó esta variável é definida com valor *falso*. Uma vez completada a subdivisão de um nó em nós filhos, o que faz com que este deixe de ser um nó externo e passe a ser um nó interno, esta variável muda seu valor de *falso* para *verdadeiro*.

6.1.1.2 A subdivisão de um nó bidimensional em nós filhos

Em nível global, há algumas constantes utilizadas para auxiliar execução e gerenciamento de uma *quadtree*, as mais importantes referem-se à profundidade máxima da árvore e a quantidade máxima de objetos que um nó pode suportar.

Sempre que a quantidade máxima de objetos por nó é extrapolada, há a necessidade da subdivisão do nó. É, então, chamado um método utilizado para subdividir o nó atual em quatro nós filhos.

Cada nó filho é um ponteiro do tipo nó que, assim como todos os objetos da classe nó, é inicializado através de um construtor que recebe por parâmetro seus limites mínimo e máximo, assim como seu nível, que trata-se do nível de seu pai

acrescido de 1. Uma vez que o construtor é chamado, ocorre o mesmo processo citado na seção anterior.

Cada nó filho possui um nome diferente pelo qual é reconhecido, estes nomes correspondem à sua posição em relação ponto central do seu nó pai. A nomeação de cada nó se dá da seguinte forma:

a) filho Sudoeste: localizado no quadrante inferior esquerdo do nó pai;

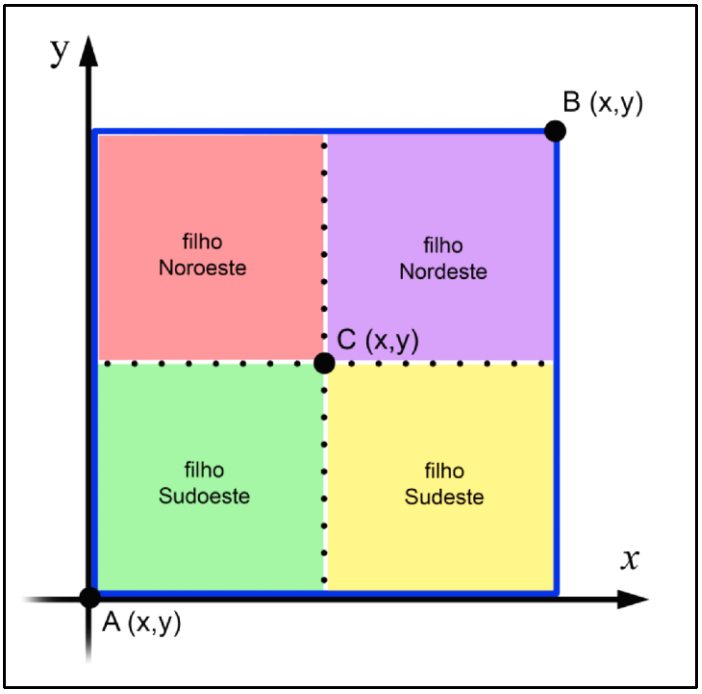
b) filho Sudeste: localizado no quadrante inferior direito do nó pai;

c) filho Noroeste: localizado no quadrante superior esquerdo do nó pai;

d) filho Nordeste: localizado no quadrante superior direito do nó pai;

Na figura 12 fica evidente o posicionamento de cada nó filho.

Figura 12 – Os filhos de uma *quadtree*



Fonte: Do autor.

Na figura 12, o ponto *A* representa o limite mínimo do nó pai, bem como *B* o seu limite máximo e *C* o seu ponto central. Estes três pontos são utilizados para definir as coordenadas mínima e máxima dos filhos Sudoeste (quadrante verde), através dos pontos *A* e *C,* e Nordeste (quadrante lilás), através dos pontos *C* e *B,* respectivamente.

Para definir os limites mínimo e máximo de cada filho, são utilizados pontos auxiliares. Estes pontos são criados e usados como argumento no momento em que o construtor é chamado. Estes argumentos serão referidos como $P_0(x,y,z)$ e

P_1 *(x,y)* e seus valores *x* e *y* são diferentes combinações dos valores *x* e *y* presentes nos limites mínimo, limite máximo e ponto central do nó pai.

O limite mínimo do filho Noroeste (quadrante vermelho) é definido por P_0 (x_A,y_C) e seu limite máximo é definido por P_1 (x_C,y_B).

O filho Sudeste (quadrante amarelo) passa por um processo semelhante: seu limite mínimo é definido através do ponto P_0 (x_C,y_A) e o seu limite máximo é definido pelo ponto P_1 (x_B,y_C).

A quantidade máxima de nós folhas que uma *quadtree* possui é igual à 4^n, onde *n* é a profundidade da árvore. Por exemplo, uma *quadtree* de profundidade igual à 3 pode possuir até 4^3 filhos, ou seja 64 nós folhas.

6.1.2 A estrutura de uma *octree*

A *octree,* assim como a *quadtree,* é uma estrutura de dados hierárquica que pode ser utilizada com a finalidade de indexação de elementos em um determinado espaço. Entretanto, ao contrário da *quadtree,* esta estrutura é responsável por trabalhar com três dimensões e não com duas.

Também representada por uma classe, uma *octree* possui todas as características fundamentais presentes em uma *quadtree.* A adição de mais um eixo, o eixo *z,* possibilita que a *octree* trabalhe com volume ao invés de área. Esta mudança influencia alguns aspectos do nó, como os pontos que representam limites mínimo e máximo, bem como o ponto central, que agora trabalham uma terceira grandeza: a profundidade.

Os métodos presentes em uma *octree* são fundamentalmente os mesmos de uma *quadtree*, dadas as devidas modificações para se trabalhar com posições tridimensionais.

6.1.2.1 A criação de uma *octree*

Com a adição de mais uma dimensão aos nós, cada nó não é mais representado por um quadrado, mas sim, por um cubo, cuja profundidade é mensurada pelo eixo *z.*

Para efetuar a criação de um nó da classe *octree* é chamado seu construtor que recebe, assim como *quadtree,* três argumentos: limite mínimo, limite

máximo e nível. Entretanto os pontos que representam os limites de um nó em uma *octree* possuem uma nova grandeza, o valor de *z.* Portanto, os pontos *A, B*, e *C* que representam, respectivamente, o limite mínimo, limite máximo e o ponto central tem seus valores definidos por *A (x,y,z)*, *B (x,y,z)* e *C (x,y,z),* sendo *x, y,* e *z,* nesta ordem, a altura, largura e a profundidade de um ponto situado no espaço, todos sendo valores numéricos com ponto flutuante.

O construtor então define os atributos espaciais do nó: o limite mínimo recebe *A,* o limite máximo recebe *B e o*s valores de *x, y* e *z* de *C* são definidos conforme nas equações (1), (2) e (3), respectivamente.

$$x_C = \frac{(x_A + x_B)}{2} \quad (1)$$

$$y_C = \frac{(y_A + y_B)}{2} \quad (2)$$

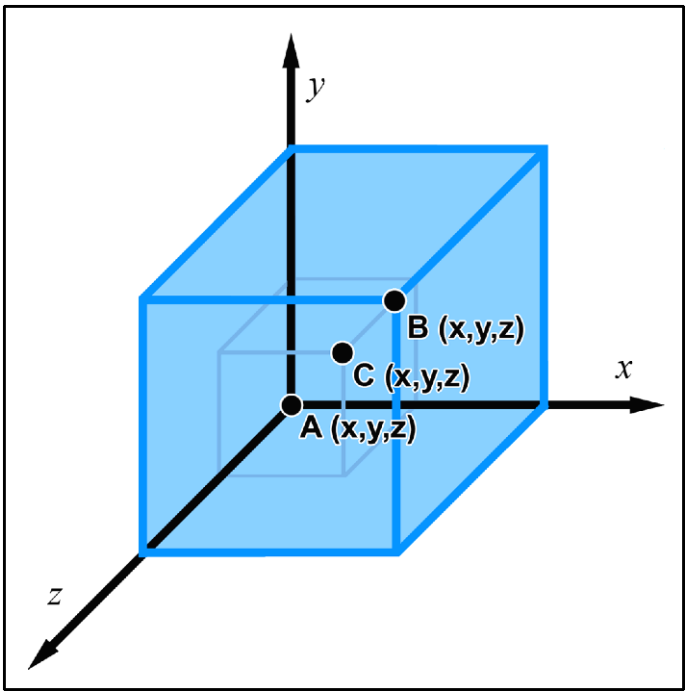
$$z_C = \frac{(z_A + z_B)}{2} \quad (3)$$

A inicialização dos demais atributos de um nó tridimensional ocorre de maneira semelhante ao que acontece na *quadtree.*

No sistema cartesiano 3D, considerando que o eixo *z* cresça em direção ao espectador, o ponto correspondente ao limite mínimo de um nó está situado no canto inferior esquerdo distante do espectador e o ponto correspondente ao limite máximo no canto superior direito próximo ao espectador.

Na figura 13, os pontos A, B e C representam, respectivamente, o limite mínimo, limite máximo e o ponto central do nó, cujo volume está em azul. O pequeno cubo formado entre *A* e *C* representa um dos 8 filhos imaginários do nó.

Figura 13 – Limites de uma *octree*



Fonte: Do autor.

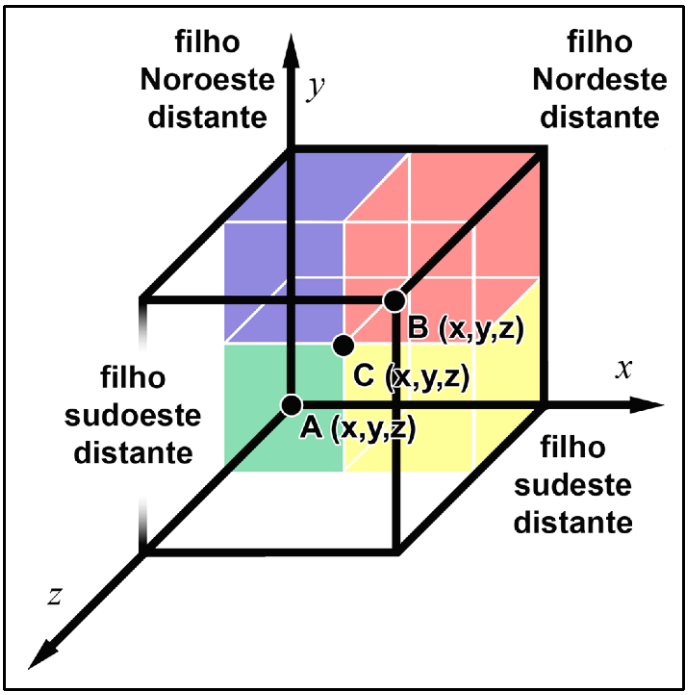
6.1.2.2 A subdivisão de um nó tridimensional em nós filhos

Assim como na *quadtree*, há variáveis globais relacionadas à quantidade máxima de objetos que um nó pode suportar e à profundidade máxima da árvore. Sempre que um nó se torna saturado, ou seja, carrega mais objetos do que lhe é permitido, é chamado um método responsável por subdividir um nó em oito nós filhos. Todos os filhos são ponteiros do tipo nó e para cada um destes é chamado um construtor de nó que recebe, seus limites mínimo, máximo e o seu nível, que é igual ao nível do seu pai mais 1.

A nomenclatura dos filhos de um nó se dá de maneira similar ao que acontece na *quadtree*: filhos Noroeste, Nordeste, Sudoeste e Sudeste. Entretanto, são divididos em duas camadas: os que estão distantes e os que estão próximos, utilizando como referência o espectador.

Na figura 14 é exemplificado o posicionamento dos nós filhos distantes quando são criados. Os três pontos mostrados na imagem são os limites mínimo, limite máximo e ponto central do nó pai, respectivamente *A, B* e C.

Figura 14 – Os filhos de uma *octree,* primeira parte



Fonte: Do autor.

Os limites mínimo e máximo de cada ponto são definidos da seguinte forma:

a) Filho Noroeste distante (em azul): limite mínimo recebe P_0 (x_A,y_C,z_A) e limite máximo recebe P_1 (x_C,y_B,z_C);
b) Filho Nordeste distante (em vermelho): limite mínimo recebe P_0 (x_C,y_C,z_A) e limite máximo recebe P_2 (x_B,y_B,z_C);
c) Filho Sudoeste distante (em verde): limite mínimo recebe *A* e limite máximo recebe *C;*
d) Filho Sudeste distante (em amarelo): limite mínimo recebe P_0 (x_C,y_A,z_A) e limite máximo recebe P_1 (x_B,y_C,z_C).

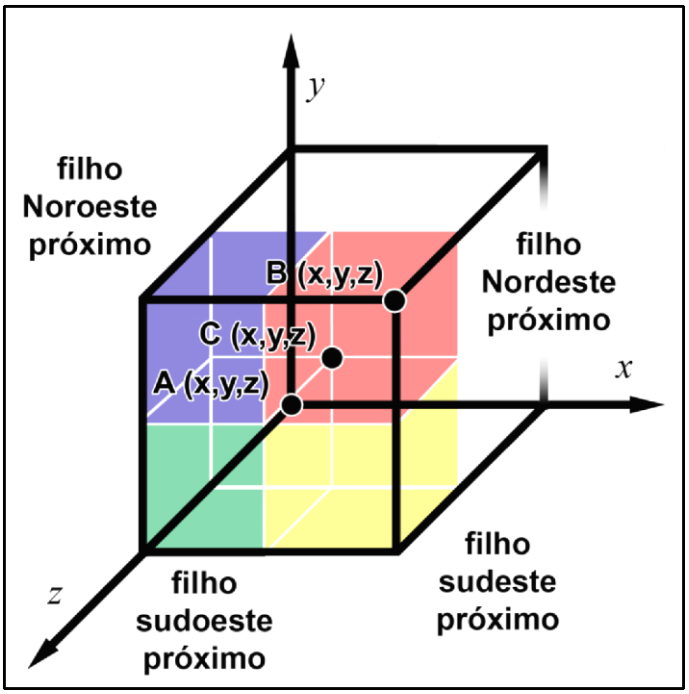
Na figura 15, é demonstrado o processo de criação dos nós filhos próximos ao espectador. Os pontos *A, B* e *C* novamente representam os limites mínimo e máximo, bem como o ponto central do nó pai e os valores *x, y* e *z* destes três pontos são combinados para determinar as dimensões de seus nós filhos.

Os nós filhos são definidos de maneira semelhante ao que acontece com seus irmãos presentes na cama distante do espectador. A definição de suas dimensões se dá da seguinte forma:

a) filho Noroeste próximo (em azul): limite mínimo recebe P_0 (x_A,y_C,z_C) e limite máximo recebe P_1 (x_C, y_B, z_B);

b) filho Nordeste próximo (em vermelho): limite mínimo recebe o ponto *C* e limite máximo recebe o ponto *B;*
c) filho Sudoeste próximo (em verde): limite mínimo recebe P_0 (x_A, y_A, z_C) e limite máximo recebe P_1 (x_C, y_C, z_B);
d) filho Sudeste distante (em amarelo): limite mínimo recebe P_0 (x_C, y_A, z_C) e limite máximo recebe P_1 (x_B, y_C, z_B).

Figura 15 - Os filhos de uma *octree*, segunda parte



Fonte: Do autor.

Para cada uma das vezes em que um construtor é chamado para inicializar as variáveis de um nó filho, ocorre o mesmo processo de criação de um nó.

A quantidade máxima de nós folhas que uma *octree* possui é igual à 8^n, onde *n* representa a profundidade da árvore. Uma *octree* de profundidade 4, por exemplo, pode possuir até 8^4 filhos, resultando em, no máximo, 4096 nós externos.

6.1.3 Inserção de objetos em quadtree e octree

Para adicionar um novo objeto à árvore é chamado um método que recebe o objeto juntamente com a posição no espaço aonde esse objeto será

adicionado. Dado que o método de inserção é semelhante em ambas as estruturas, este será abordado de maneira genérica.

A posição espacial do objeto a ser inserido pode ser bidimensional, no caso da *quadtree* ou tridimensional, no caso da *octree.*

Quando o método de inserção é executado, o objeto é adicionado à lista de objetos do nó correspondente, a quantidade de objetos do nó em questão é aumentada em 1 e, logo em seguida, é efetuado um teste para checar se a quantidade de objetos do nó é maior ou igual à quantidade máxima de objetos permitida por nó.

Se o teste retornar *verdadeiro* e o nó não possuir filhos, é chamada a função responsável por subdividir o nó em nós filhos: quatro para uma *quadtree* e oito para uma *octree.* Após a subdivisão, todos os objetos que o nó possui, incluindo o novo objeto recém inserido, são classificados com o auxílio de um método responsável por localizar o objeto em relação aos nós filhos para que sejam, então, adicionados no filho correspondente à sua localização.

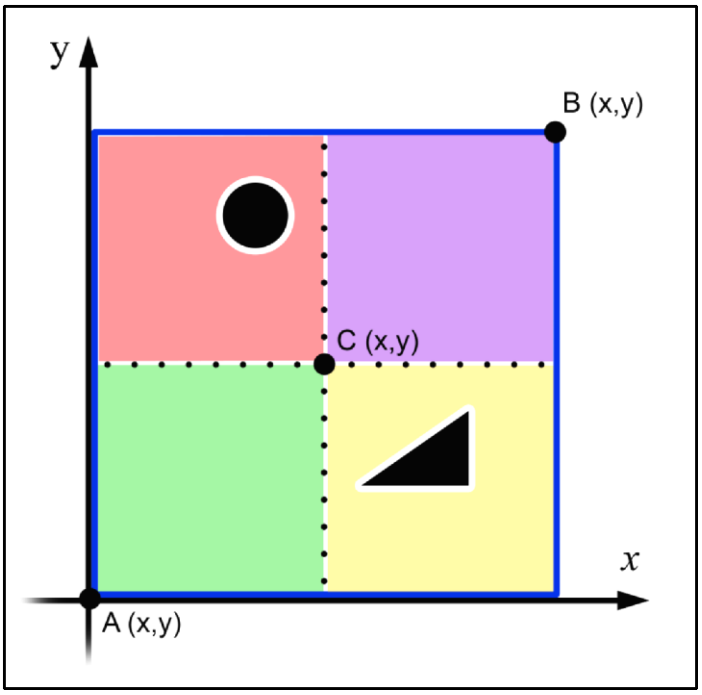
Se o teste retornar *verdadeiro* e o nó possuir filhos, é chamado o método responsável pela localização para que o objeto seja inserido, em algum dos nós filhos de acordo com seu posicionamento.

Porém, se o primeiro teste retornar *falso*, o que significa que o número de objetos atual do nó não supera nem se iguala à quantidade máxima de objetos permitida por nó, o objeto é apenas adicionado à lista de objetos do nó.

Na figura 16, há um exemplo de nó, cujas arestas estão em azul. Este nó, devido a inserção de mais um objeto, teve seu limite de objetos por nó (neste exemplo, 1) extrapolado e devido à isso passou pelo processo de subdivisão, de modo que, agora, possui quatro nós filhos, cujas áreas são representadas pelos quadrantes vermelho, lilás, verde e amarelo.

Entretanto, os objetos não podem continuar no nó pai, portanto eles são transferidos do nó pai para uma lista auxiliar e então para cada um é chamado o método de localização que testará a posição de cada objeto individualmente de acordo com os limites de cada quadrante, utilizando o ponto central como referência.

Figura 16 – Inserção de novos objetos em uma *quadtree*



Fonte: Do autor.

A figura 16 representa o pseudo-código do método responsável por localizar um objeto em uma *quadtree*. As linhas 3, 5, 6 e 10 representam a chamada do método responsável pela adição de objeto ao quadrante no qual se adequa.

Em uma *octree,* o processo de localização ocorre de maneira semelhante, de modo que os parâmetros comparativos utilizados para localizar o objeto também utilizam o eixo *z.*

Figura 17 – Pseudo-código do método de localização de objetos de uma *quadtree*

```
 1  SE y_objeto < y_centro ENTÃO
 2      SE x_objeto < x_centro ENTÃO
 3          filho_sudoeste->adicionar_objeto(objeto);
 4      SENÃO
 5          filho_sudeste->adicionar_objeto(objeto);
 6  SENÃO
 7      SE x_objeto < x_centro ENTÃO
 8          filho_noroeste->adicionar_objeto(objeto);
 9      SENÃO
10          filho_nordeste -> adicionae_objeto(objeto);
11
```

Fonte: Do autor.

Uma vez que árvores possuem natureza recursiva, a transferência de objetos de um nó para seus filhos ocorre da mesma forma. Se, no momento em que um objeto for adicionado à um nó, a quantidade de objetos deste nó já tiver sido

extrapolada, o método de localização é chamado para que possa localizar o objeto e, em seguida, chamar o método de inserção de objetos para o filho correspondente à posição do objeto.

Em todos os testes acima citados, há a necessidade de se fazer uma verificação quanto à profundidade máxima que a árvore pode atingir. Se a profundidade máxima já foi alcançada, o objeto é adicionado ao nó, ignorando a regra de número máximo de objetos por nó; do contrário, o processo de transferência e inserção de objetos ocorre recursivamente até que a profundidade máxima seja alcançada ou que todos os objetos sejam distribuídos entre os nós da árvore de maneira que nenhum nó possua uma quantidade de objetos maior ou igual ao número máximo permitido por nó.

6.1.4 A remoção de objetos e nós

O método relacionado à remoção de objetos no espaço, seja 2D ou 3D, no momento em que é chamado, recebe como argumento a posição de um objeto existente. O método então percorre a árvore, guiando-se através do método responsável pela localização de objetos, de modo que o objeto utilizado como argumento na função de remoção sirva como referência.

No momento em que o nó no qual o argumento deveria estar localizado é encontrado, é efetuado um laço, comparando todos os objetos contidos no nó com o objeto argumento para verificar se há algum cujas características se encaixem.

Se o teste retornar *verdadeiro*, o objeto é removido da lista de objetos daquele nó e o a quantidade de objetos daquele nó é reduzida em 1; se *falso* o método é apenas finalizado.

No momento em que um nó é removido, é executado um segundo método responsável pela limpeza e atualização da árvore. Este método percorre, recursivamente, todos os nós da árvore e destrói todos os nós filhos cujos pais possuam uma quantidade de objetos igual a 0.

6.1.5 A detecção de colisão

A detecção de colisão entre objetos no espaço consiste em verificar e retornar se, a cada iteração do programa, há a intersecção da área ou volume de dois ou mais objetos no espaço.

O método tradicional para se detectar colisão entre polígonos no espaço trata-se de um método que usa força bruta, ou seja: a cada iteração todos os objetos no espaço são testados contra todos os objetos no espaço, de modo a verificar se há intersecção entre um ou mais objetos e retornar essas intersecções a fim de tratá-las. A quantidade de testes necessários por iteração é definida por:

$$\sum_{i=1}^{n-1} i = 1 + 2 + 3 + \cdots + (n-1)$$

No somatório acima, *n* representa a quantidade de objetos a serem testados no espaço. Portanto, num espaço em que há 8 objetos se movendo, a quantidade necessária de testes por iteração é igual à 28.

O método que se utiliza de estruturas de dados para indexar o espaço, bem como os objetos neste presentes, trabalha de uma forma diferente. Ao invés de o somatório se aplicar uma única vez para o valor total de objetos, ele se aplica diversas vezes para cada uma das folhas da árvore que possua objetos dentro de seus limites, dividindo o problema em problemas menores e resultando numa menor quantidade de testes.

Isso se dá porque porque não há necessidade de testar todos os objetos contra todos os objetos. A detecção de colisão em um espaço indexado ocorre de forma seleta, realizando testes apenas onde há prováveis colisões, justamente porque não há razão para testar uma possível colisão entre objetos presentes em nós fisicamente distantes.

Essa filtragem gera uma redução significativa na quantidade de testes efetuados por iteração e o ganho computacional frente ao método de força bruta fica mais evidente conforme a quantidade de objetos situados no espaço aumenta.

6.2 O ALGORITMO UTILIZADO

O sistema utilizado para realizar os testes, trata-se de uma adaptação de um algoritmo já existente. Este algoritmo, criado por Bill Jacobs e publicado em sua *homepage*, com propósitos educacionais, foi desenvolvido na linguagem de programação C++ com a utilização da Interface de Programação de Aplicativos, do inglês *Application Programming Interface* (API), OpenGL.

O algoritmo em si é parte de uma lista de tutoriais desenvolvidos com a finalidade de ensinar OpenGL utilizando uma abordagem focada à programação e seu código-fonte possui licença livre, sem qualquer limitação para uso, cópia, modificação, publicação, distribuição e/ou venda. Jacobs, entretanto, deixa claro que se isenta de toda e qualquer responsabilidade relacionada ao uso do seu algoritmo.

6.2.1 A *octree* de Bill Jacobs

O código-fonte disponibilizado por Bill Jacobs em sua *homepage,* trata-se de uma *octree* e é utilizado em algumas de suas vídeo-aulas para explicar o conceito de indexação espacial, bem como demonstrar uma solução eficiente para realizar detecção de colisão entre objetos no espaço.

Os objetos utilizados por Jacobs são esferas, inseridas aleatoriamente no espaço e que se movem, também, de maneira aleatória. A cada iteração, um método percorre a árvore em busca de esferas que estejam no mesmo nó e as armazena em uma lista que relaciona as esferas em pares, definindo-as como possíveis colisões. Deste ponto em diante, são efetuados testes, par por par, de modo a verificar se as esferas estão ou não interesectando o volume uma da outra.

Se, durante a verificação, alguma colisão for detectada, esta colisão é então tratada através de um método que simula a reação física derivada da colisão das esferas.

Em seu algoritmo, Jacobs, define as esferas como uma *struct* que possui raio, definido por um valor com ponto flutuante, velocidade, cor e posição; estes três últimos valores definidos pelo tipo *vec3f*, um tipo construído desenvolvido por ele próprio e implementado como uma biblioteca que é incluída no cabeçalho do código-fonte.

Uma variável do tipo *vec3f* é definida por um vetor com três elementos, todos com ponto flutuante e pode ser entendida como um ponto *A* que possui três grandezas: *x, y* e *z*. No caso da velocidade, os valores de *A* representam a velocidade de deslocamento em *x, y* e *z;* para o atributo cor, representam os valores de vermelho, verde e azul e para atributo posição, representam a posição da esfera em relação aos eixos *x, y* e *z.*

A estrutura da *octree* e a maneira como se comporta frente ao gerenciamento dos objetos que possui é essencialmente a mesma abordada nas seções anteriores, salvo pequenas exceções. Os nós filhos, por exemplo, são representados por um vetor tridimensional de ponteiros do tipo *octree* com largura, altura e profundidade iguais à 2, de modo a gerar 8 combinações possíveis. A utilização e gerenciamento de nós filhos desta forma otimiza o deslocamento dos métodos através dos elementos da árvore, de modo que os filhos de um nó podem ser facilmente acessados se conhecida a sua posição dentro do vetor.

As esferas se movem no espaço com o auxílio do atributo velocidade. A cada iteração é incrementado aos valores *x, y* e *z* da sua posição atual os valores *x, y,* e *z*, respectivamente, do seu atributo velocidade. Quando algum dos eixos do atributo velocidade de uma esfera é igual a 0, significa que a esfera não se move naquele eixo. Por exemplo, uma esfera que possua o atributo velocidade como *vec3f (2,3,0)* a cada iteração se move 2 unidades em *x*, 3 unidades em *y* e nenhuma unidade em *z.*

6.3 SISTEMA DESENVOLVIDO

O objetivo deste trabalho é tornar evidente a diferença de eficiência computacional ao realizar detecção de colisão entre objetos situados no espaço bidimensional e tridimensional utilizando-se de técnicas que utilizam estruturas de dados hierárquicas e técnicas convencionais.

O código-fonte da *octree* desenvolvida por Jacobs foi utilizado como base para o desenvolvimento dos ambientes utilizados para testes no espaço bidimensional e tridimensional e toda a parte que envolve programação foi desenvolvida no ambiente de desenvolvimento CodeBlocks utilizando a API OpenGL, dado que ambas são alternativas gratuitas.

O desenvolvimento da *quadtree* consistiu-se do desmembramento da *octree* de modo a remover uma de suas dimensões. Para tal, foi necessário a remoção de 4 dos seus filhos, tornando o vetor de ponteiros tridimensional responsável pela alocação dos filhos em um vetor de ponteiros bidimensional, bem como a adaptação de alguns de seus métodos para trabalhar com nós bidimensionais ao invés de pontos tridimensionais.

Os pontos relacionados aos limites mínimo e máximo, assim como o ponto central, foram conservados como pontos tridimensionais porém, com o valor relacionado ao eixo *z* sempre inicializado como 0, de modo tornar o espaço em um plano, impedindo que objetos desloquem-se no eixo *z.*

Para a detecção de colisão no espaço bidimensional através do método tradicional, foi utilizado como base o algoritmo da *quadtree* e foram preservados apenas os atributos relacionados ao espaço no qual os testes serão efetuados que, nada mais é, do que um nó raiz sem filhos. Os métodos relacionados ao gerenciamento de nós filhos foram removidos; já os demais métodos, relacionados a manutenção, inserção e remoção de objetos no espaço foram preservados.

O ambiente de detecção de colisão no espaço tridimensional surgiu à partir da *octree,* do mesmo modo que o ambiente de detecção de colisão no espaço bidimensional surgiu da *quadtree,* dadas as devidas proporções. Foram removidos os nós filhos e os métodos responsáveis pelo gerenciamento de nós filhos, entretanto, todos os outros métodos foram preservados.

A *octree* foi a estrutura que sofreu a menor quantidade de alterações, recebendo apenas os métodos responsáveis por representar graficamente os nós, algumas linhas de código inseridas para realizar as medições de tempo e eficiencia, alterações estas, também inseridas em todos os ambientes anteriores.

Para a implementação do método responsável por desenhar os nós na tela, foi necessária a criação de um método *getter* para cada um dos nós filhos de cada nó, bem como para cada um dos atributos do tipo construído *vec3f.*

O método responsável por desenhar os nós na tela é chamado a cada iteração e desenha todas as arestas de cada nó. Começando pelo nó raiz, ele desenha nó raiz, então verifica se este possui filhos, se sim, então o método é chamado novamente para cada um dos filhos e assim recursivamente. Até que tenha desenhado todos os nós da árvore.

A medição de tempo foi realizada com a utilização de duas variáveis: a variável *inicio* que recebe o tempo em que método *handleBallBallCollisions,* responsável pela detecção de colisão, começa a ser executado e a variável *fim* que recebe o tempo em que o método termina sua execução. A diferença entre *fim* e *inicio* representa a duração, em milisegundos, da execução método *handleBallBallCollisions*. A cada iteração este valor de tempo obtido é acumulado em uma terceira variável, de modo que, no final da execução do programa, se obtenha o tempo total relacionado apenas à detecção de colisão e não a outras rotinas, como a de desenhar os nós na tela, por exemplo.

Também foi inserida uma variável que, sempre que é realizado um teste para verificar se duas esferas contidas em um par se intersectam, incrementa seu valor em 1. Esta variável, ao final da execução do programa conterá a quantidade total de testes executados durante as iterações.

Em todos os quatro casos foi comentado o trecho de código que simula a aceleração da gravidade pois foi constatado que, ao invés dos objetos gradualmente perderem velocidade e tenderem ao repouso, o contrário acontecia: sua velocidade crescia de maneira indefinida.

Dada também a constatação de que as colisões entre as esferas e os limites do nó aconteciam de maneira imprecisa, estas foram desconsideradas, de modo que seu tempo de execução e seus testes de colisão não são contabilizados.

6.4 TESTES REALIZADOS

Para a realização das simulações em todos os quatro ambientes foi definido um valor fixo de iterações igual à 128. No momento em que a 128ª iteração é finalizada o programa termina sua execução e imprime na tela o tempo total utilizado para realizar detecção de colisão ao longo das 128 iterações, sendo este o tempo efetivo utilizado para a execução da detecção de colisão; o total de testes efetuados, considerando que são contabilizados apenas os testes realizados com pares de esferas; uma média de testes por iteração, que consiste na quantidade total de testes dividida pelo número de iterações; e o total de colisões detectadas.

Para as estruturas de dados hierárquicas foi definida uma profundidade máxima igual à 3 e uma quantidade máxima de esferas por nó igual à 2 como critério para subdivisão do nó em nós filhos.

Foram realizados testes com 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 4096 e 8192 esferas e foram realizados cinco testes para cada uma das quantidades de esferas, de modo a se obter uma média.

6.5 RESULTADOS OBTIDOS

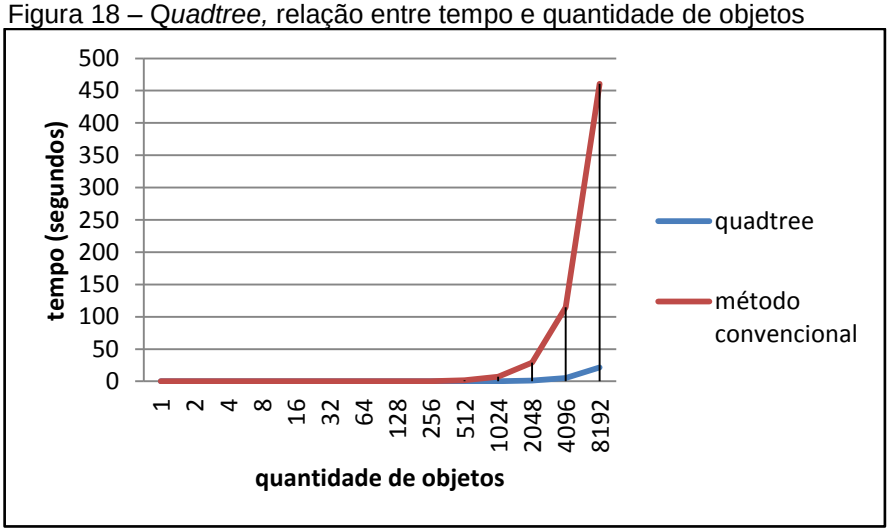
Durante a realização dos testes foram encontrados alguns contratempos. Observou-se que a impressão de valores no *prompt* de comando durante a execução de cada teste, influenciava no tempo total de cada iteração. Devido à isto, este recurso que inicialmente imprimia o atual estado de cada uma das iterações foi removido de modo que os testes pudessem sem realizados da melhor maneira possível, visando ter uma medida de tempo final o mais próxima do real.

O programa se comportou da maneira esperada: realizando todas as 128 iterações adequadamente e imprimindo os resultados na tela posteriormente. Estes resultados foram utilizados para criar gráficos de modo a expressar com mais clareza a comparação entre a eficiência da cada algoritmo.

Para a elaboração das representações gráficas dois critérios foram avaliados: o tempo total de duração da execução de todos os testes relacionados à detecção de colisão e a quantidade total de testes realizada ao final da execução do programa.

Na figura 18, fica claro como a quantidade de objetos influencia o tempo final resultante da soma da execução de todos os testes. Inicialmente, até o ponto em que são testados 512 objetos, ambos os algoritmos tem um tempo de execução próximo à 0, à partir deste ponto o método convencional começa a crescer de maneira exponencial, dado que, ao testar uma quantidade de 8192 objetos o método convencional requer aproximados 450 segundos para executar todos os testes contra menos de 25 segundos necessários para a *quadtree* realizar todos os cálculos.

Figura 18 – Q*uadtree,* relação entre tempo e quantidade de objetos

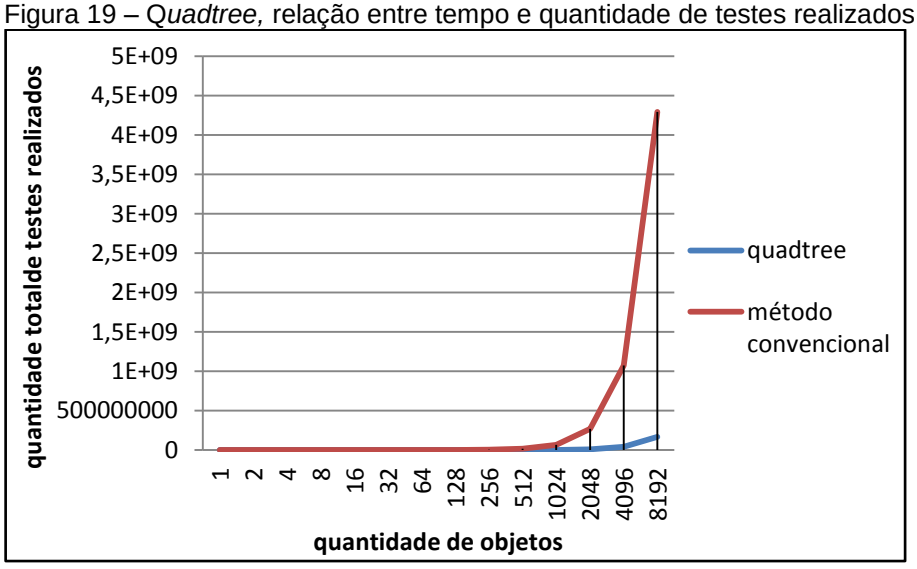


Fonte: Do autor.

Esta diferença se dá, principalmente, pelo fato de a quantidade de testes necessárias por iteração crescer junto com a quantidade de objetos presentes no espaço.

Dada a figura 19, que relaciona a quantidade de testes total ao final da execução do programa com a quantidade de objetos presentes no espaço, é possível perceber que o aumento de tempo presente no gráfico da figura 18 está diretamente relacionando com a quantidade total de testes realizada no término da execução do programa. Para uma quantidade de objetos igual à 8192 a quantidade de testes contabilizados ao final da execução do programa é um valor próximo à 4,5 bilhões enquanto a *quadtree* executa cerca de 160 milhões de testes.

Figura 19 – Q*uadtree,* relação entre tempo e quantidade de testes realizados



Fonte: Do autor.

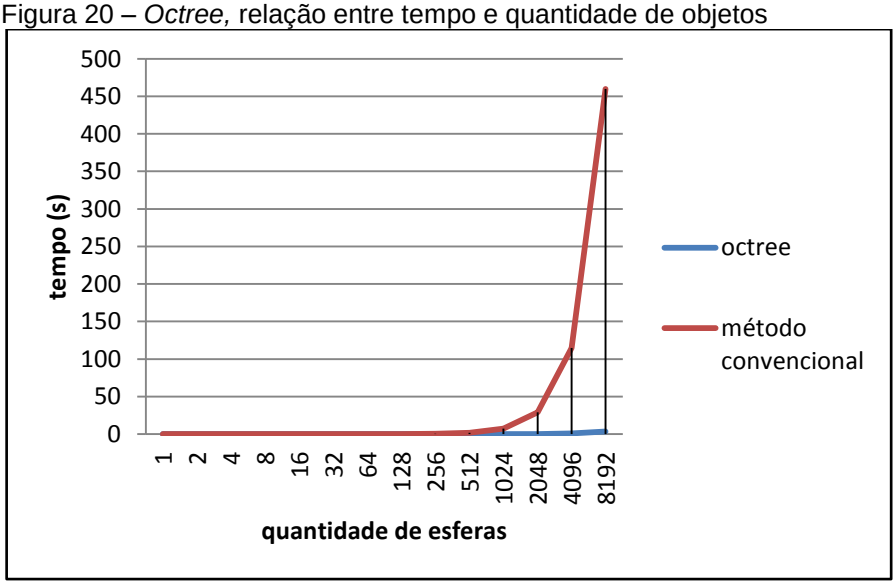
Haja visto uma *quatree* reduz significantemente a quantidade de testes necessários para detectar colisão, testando apenas objetos vizinhos. É visível o ganho de desempenho frente ao método de força bruta que, inicialmente, possui o mesmo desempenho que a *quadtree.*

Nos testes realizados no espaço tridimensional os resultados foram similares: ambos os métodos, nos primeiros testes, mantém um crescimento estável e desempenho semelhante, porém, quando a quantidade de objetos cresce para 512 objetos o tempo de execução total dos testes do método convencional começa a crescer exponencialmente, assim como ocorre no ambiente bidimensional.

A figura 20 mostra a relação o tempo e a quantidade de objetos presente nas várias execuções dos ambientes tridimensionais. Assim como no sistema bidimensional foi constatado um crescimento exponencial do tempo de execução das detecções de colisão por parte do método convencional.

Foi observado, entretanto, que a *octree* obteve um desempenho maior do que a *quadtree.* Isto não se dá pelo fato de a *octree* ser mais eficiente do que uma *quatree*, mas ao fato de que o espaço para os objetos transitarem no espaço tridimensional é maior, o que resulta em uma quantidade menor de possíveis colisões, resultando assim, no notável aumento de desempenho.

Figura 20 – *Octree,* relação entre tempo e quantidade de objetos

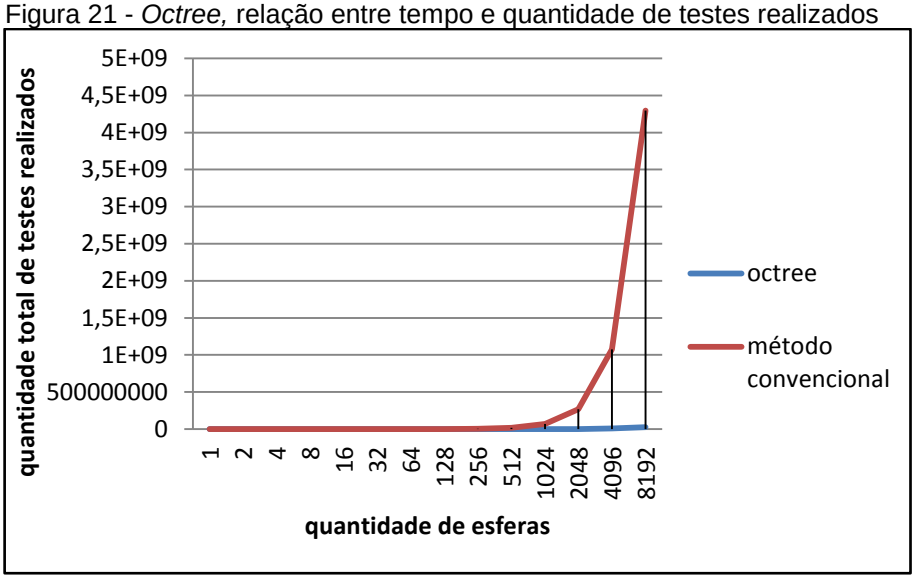


Fonte: Do autor.

Ao se analisar a figura 21, assim no caso do ambiente bidimensional, é possível realizar uma ligação entre aumento do tempo total e a quantidade de testes

necessários para cada caso. Ambos crescem de maneira exponencial a partir do momento em que há mais de 512 objetos se movendo no espaço.

Figura 21 - *Octree,* relação entre tempo e quantidade de testes realizados



Fonte: Do autor.

7 CONSIDERAÇÕES FINAIS

Dadas diferentes situações e ambientes as estruturas de dados hierárquicas *quadtree* e *octree* se mostraram uma solução viável para indexação de eventos espaciais, demonstrando uma perda mínima de desempenho, frente ao método convencional que utiliza força bruta para realizar a detecção de colisão.

Visto que possuem uma natureza maleável, de modo que podem ser esculpidas para atender os mais diversos fins, estas estruturas de dados se mostraram uma solução robusta, precisa e eficiente para realizar e gerenciar detecção de colisão em ambientes bidimensionais e tridimensionais.

O estudo realizado ao longo deste trabalho pode ser utilizado como ponto de partida para o estudo e criação de uma técnica de tratamento de colisão. O estudo do tratamento de colisão abre uma gama de possibilidades pra simulações avançadas como simulação de partículas, liquidos ou malhas, por exemplo.

REFERÊNCIAS

ANGEL, E. **Interactive Computer Graphics - A Top-Down Approach with OpenGL**. 2. ed. Reading: Addison-Wesley, 2000.

ALMEIDA, Vitor Pinheiro de. **Detecção de colisão através de Octree**. Disponível em: <http://www.puc-rio.br/pibic/relatorio_resumo2007/resumos/MAT/vitor_pinheiro_almeida.pdf> Acesso em: 5 jun 2011.

AZEVEDO, Eduardo; CONCI, Aura. **Computação gráfica: teoria e prática**. Rio de Janeiro: Elsevier; Campus, 2003.

BLACK, Paul E. **Dictionary of Algorithms and Data Structures.** Disponível em <http://xlinux.nist.gov/dads/> Acesso em: 22 nov 2011.

DROZDEK, Adam. . **Estrutura de dados e algoritmos em C++.** São Paulo: Thomson, 2002

EDER, J. **Octree**. Disponível em: <http://www.cg.tuwien.ac.at/studentwork/VisFoSe98/eder/octree.htm> Acesso em: 1 jun 2011.

FOLEY, James D. **Computer graphics: principles and practice**. 2nd ed. California: Addison-Wesley, 1995.

FOLEY, James D. **Introduction to computer graphics**. New York, USA: Addison-Wesley, 2000.

GONZALEZ, Rafael C.; WOODS, Richard E. **Processamento de imagens digitais**. São Paulo: Edgard Blücher, 2000.

JACOBS, Bill. **OpenGL Tutorial.** Disponível em: <http://www.videotutorialsrock.com/index.php> Acesso em 22 jan 2012.

KASTELIE, Willian P. **The Pair tree: a parallel architecture for image representation based on symmetric recursive indexing.** Canada: Simon Fraser University, 1989.

KOFFMAN, Elliot B.; WOLFGANG, Paul A. T. **Objetos, abstração, estruturas de dados e projeto usando C++**. Rio de Janeiro: LTC, 2008.

MARROQUIM, Ricardo G. **Detecção de Colisão.** Disponível em <http://www.lcg.ufrj.br/Cursos/jogos/04-deteccao-colisao.pdf> Acesso em 21 nov 2011.

MENDES, Vilson B. **Noções Básicas sobre Métodos de Modelagem.** Disponível em < http://www.ic.uff.br/~aconci/model.htm> Acesso em: 23 nov 2011.

MILANEZ, Bruno Abel**. Interpolação Aplicada À Animação Gerada Por Computador**. Criciúma, SC: UNESC, 2009. Trabalho de Conclusão de Curso (Bacharel em Ciência da Computação) – Universidade do Extremo Sul Catarinense.

NOVAK, Jeannie. **Desenvolvimento de games**. Tradução Pedro Cesar de Conti; revisão técnica Paulo Marcos Figueiredo de Andrare. 2nd ed. São Paulo: Cengage Learning, 2010.

NYHOFF, Larry R. **ADTs, data structures, and problem solving with C++.** 2nd ed Upper Saddle River, NJ: Prentice Hall, 2005.

PENTON, Ron. **Data Structures For Game Programmers.** Ohio: Premier Press, 2003.

PREISS, Bruno R. **Data Structures and Algorithms with Object-Oriented Design Patterns in C++.** Disponível em <http://www.brpreiss.com/> Acesso em 22 nov 2011.

QUADTREE. In: Wikipédia: A Enciclopédia Livre. Disponível em: <http://en.wikipedia.org/wiki/Quadtree> Acesso em: 22 nov 2011.

SHERROD, Allen. **Data structures and algorithms for game developers.** Massachussetts: Charles River Media, 2007.

SUTER, Jaap. **Introduction to Octrees**. Disponível em: <http://www.flipcode.com/archives/Introduction_To_Octrees.shtml> Acesso em: 17 mai 2011.

WAGNER, Harley. **Quadtrees e Octrees.** Disponível em: **<**http://www.inf.ufsc.br/~visao/1998/harley/octree.htm**>** Acesso em: 29 jun 2012.

XIANG, Zhigang; PLASTOCK, Roy A. **Schaum's Outline of Theory and Problems of Computer Graphics**. Ohio: McGraw-Hill Professional, 2000.

_____. **Gamma: Geometric Algorithms for Modeling, Motion and Animation**. Disponível em: <http://gamma.cs.unc.edu/> Acesso em: 1 jun 2011.

ANEXO(S)

Detecção de Colisão em ambientes Bidimensionais e Tridimensionais Utilizando Estruturas de Dados Hierárquicas Quadtrees e Octrees

Jonas Gabriel de Souza[1], Evânio Ramos Nicoleit[2]

[1]Acadêmico do Curso de Ciência da Computação – Universidade do Extremo Sul Catarinense (UNESC)
Criciúma – SC – Brazil

[2]Professor do Curso de Ciência da Computação – Universidade do Extremo Sul Catarinense (UNESC)
Criciúma – SC – Brazil

johnthebatbiter@gmail.com, evanio@unesc.net

Abstract. *To detect collisions among objects and other events in the environment is a considerable problem in the simulation area, since the computer resources required grow in a proportional scale to the quality to be achieved. Therefore, when it comes to quality and performance, it is important to pay attention when choosing the right technic to do such tasks. However, it is possible to set up the balance between power and efficiency using quadtrees and octrees. These hierarchical data structures combine both characteristics, performance and quality, so they are able to attend to many needs. Showing itself as a flexible and, at the same time, efficient solution to the spacial events indexing.*

Resumo. *A detecção de colisão entre objetos e outros eventos situados no espaço é uma grande problemática nas áreas da simulação, uma vez que o custo computacional cresce de maneira proporcional à complexidade computacional envolvida. Portanto, a escolha da técnica utilizada para realizar estas tarefas se torna importante, tanto em questão de qualidade quanto de desempenho. Entretanto, é possível encontrar este equilíbrio entre robustez e eficiência com a utilização de quadtrees e octrees. Estas estruturas de dados hierárquicas, combinam desempenho e qualidade e podem atender à diversas necessidades, tornando-se uma solução flexível e ao mesmo tempo eficiente para a indexação de eventos no espaço.*

1. Introdução

Segundo Foley (1995), o uso da Computação Gráfica tem se tornado cada vez mais popular para as ciências e engenharias. Pode-se usar tal recurso para representar modelos matemáticos de fenômenos tais como simulação de fluidos, relatividade, reações químicas e nucleares, sistemas fisiológicos e funções de órgãos, deformação de estruturas mecânicas em reação à vários tipos de pressão, dentre outros. Sendo assim, a Computação Gráfica tornou-se indispensável no campo da simulação, pesquisa e desenvolvimento das mais diversas áreas.

Com a necessidade de simulações físicas cada vez mais precisas, sejam para a simulação, entretenimento e afins, algoritmos eficientes para determinadas tarefas são cada vez mais exigidos. Neste contexto, a detecção de colisão tem sido um problema fundamental na animação por computadores, modelagem física, modelagem geométrica e robótica. Nestas aplicações, interações entre objetos que se movem são modeladas com limites dinâmicos e análise de contato.

Uma colisão é definida pela interesecção da área ou volume de dois ou mais artefatos situados no espaço, de modo que a movimentação destes objetos em uma situação de simulação deve ser restrita por várias interações, incluindo as colisões.

A solução para esta problemática é encontrada na indexação espacial. Há inúmeras estruturas de dados para representação e indexação de eventos no epaço e que podem ser aplicadas à detecção de colisão entre objetos. Estruturas de dados hierárquicas possibilitam testes recursivos nas regiões representadas, muitas vezes reduzindo a complexidade computacional envolvida.

Esta tarefa pode ser realizada por meio de estruturas de dados hierárquicas chamadas *quadtrees*, que trabalham com o espaço bidimensional, e *octrees*, que tralham com o espaço tridimensional. Tais estruturas possuem natureza maleável, de modo a torná-las uma alternativa eficiente para indexar e detectar eventos no espaço.

De modo a evidenciar a eficiência destas estruturas de dados, este artigo aborda um estudo comparativo realizado entre as mesmas e o modo convencional de detecção de eventos no espaço com foco na detecção de colisão e em uma análise quantitativa dos resultados obtidos.

2. Detecção de colisão

Uma colisão acontece no momento em que há a intersecção da área ou volume de dois ou mais objetos situados no espaço. A detecção de tal intersecção acontece através de cálculos que, em meio computacional, assume que os artefatos situados no espaço possuam a propriedade de sólidos.

Conforme Marroquim (2011), os problemas frequentemente encontrados nos processos de detecção de colisão estão relacionados a velocidade dos objetos no espaço, a complexidade geométrica destes objetos, técnicas para detecção de colisão em sistemas com prioridade de tempo real ou ao alto custo computacional necessário para que a detecção seja satisfatoriamente realizada.

Dados estes problemas, se torna evidente a importância de se encontrar o equilíbrio entre qualidade e desempenho, de modo que a solução para o problema seja robusta e ao mesmo tempo eficiente. Este equilíbrio é encontrado em estruturas de dados chamadas *quadtree* e *octree* que, dentre muitas outras finalidades, podem ser utilizadas para indexar eventos no espaço.

3. *Quadtrees, Octrees* e o método tradicional de detecção de colisão

Quadtrees são um tipo de estrutura de dados hierárquica, comumente chamada de *árvore*, na qual cada elemento possui exatamente 4 filhos. São utilizadas como uma técnica para se representar o espaço de modo a possibilitar a reprodução de imagens bidimensionais. Para se obter uma *quadtree* é necessária a delimitação de uma área no espaço bidimensional e então subdividí-la sucessivamente, de modo a formar quadrantes [Mendes 2011].

Estas árvores podem ser usadas para diversas funções como indexar a área de um objeto no plano ou armazenar as cores que cada pixel possui em uma imagem. Independente da tarefa, sua estrutura é a mesma. Seus nós podem carregar diferentes tipos de dados e a precisão de uma *quadtree* é determinado pela quantidade de níveis que a árvore possui.

Para que seja possível realizar de detecção de colisão com *quadtree* é necessário realizar primeiramente a indexação espacial da área na qual se deseja detectar colisões. A indexação espacial ocorre no momento em que a estrutura de dados é criada e, durante este

processo, a estrutura recursivamente verifica há a presença de elementos no espaço em que está situada e, recursivamente, os separa entre seus nós filhos.

Para tal feito, são utilizados métodos recursivos e uma abordagem *top-down*. O processo de subdivisão de um quadrante em quadrantes menores acontece sucessivamente até que uma condição, arbitrariamente definida, seja saciada. No caso da Figura 1, esta condição refere-se à cada nó carregar apenas um círculo como informação.

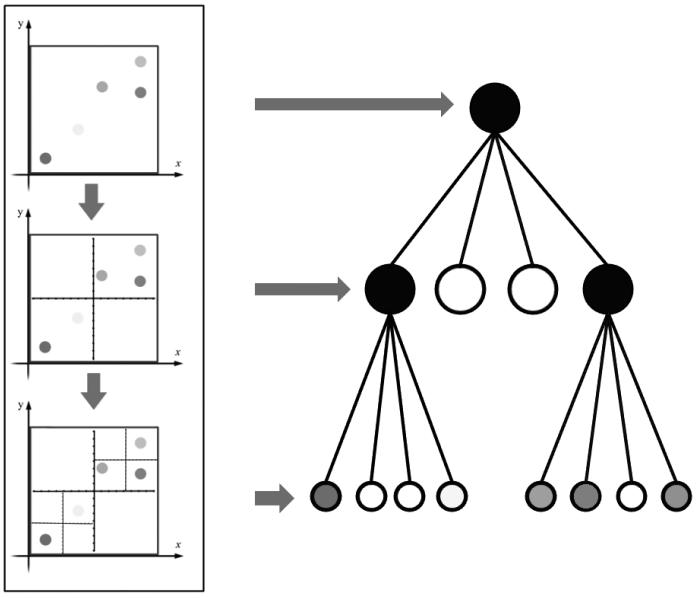


Figura 1. Indexação espacial com *quadtree*

Esta subdivisão dos objetos entre os quadrantes é o ponto de partida para realizar detecção de colisão utilizando esta estrutura de dados. Ao invés de serem testadas todas as possíveis intersecções, são apenas testados os objetos que estão nos quadrantes vizinhos. Isso reduz significantemente a quantidade de testes de colisão necessários por iteração.

Assim como a *quadtree*, a *octree* também é utilizada para representar espaço. Entretanto, ao contrário da estrutura abordada anteriormente, a *octree* é utilizada para representar espaço tridimensional, ou seja: volume. Esta mudança implica na adição de mais filhos por nó, de modo que cada elemento da árvore, caso possua, terá 8 filhos ao invés de apenas 4.

Se processo de criação da árvore e indexação do espaço é semelhante ao da *quadtree*, porém, com a adição de mais um eixo no espaço (figura 2).

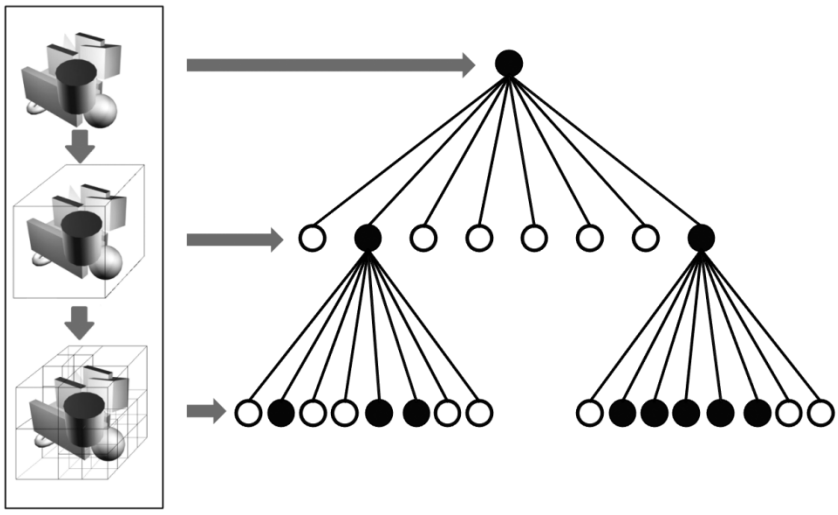


Figura 2. Indexação espacial com *octree*

O método tradicional para se detectar colisão entre polígonos no espaço trata-se de um método que usa força bruta, ou seja: a cada iteração todos os objetos no espaço são testados contra todos os objetos no espaço, de modo a verificar se há intersecção entre um ou mais objetos e retornar essas intersecções a fim de tratá-las. A quantidade de testes necessários por iteração é definida por:

$$\sum_{i=1}^{n-1} i = 1 + 2 + 3 + \cdots + (n-1)$$

Figura 3. Quantidade de testes por iteração

No somatório acima, *n* representa a quantidade de objetos a serem testados no espaço. Portanto, num espaço em que há 8 objetos se movendo, a quantidade necessária de testes por iteração é igual à 28.

O método que se utiliza de estruturas de dados para indexar o espaço, bem como os objetos neste presentes, trabalha de uma forma diferente. Ao invés de o somatório se aplicar uma única vez para o valor total de objetos, ele se aplica diversas vezes para cada uma das folhas da árvore que possua objetos dentro de seus limites, dividindo o problema em problemas menores e resultando numa menor quantidade de testes.

Isso se dá porque porque não há necessidade de testar todos os objetos contra todos os objetos. A detecção de colisão em um espaço indexado ocorre de forma seleta, realizando testes apenas onde há prováveis colisões, justamente porque não há razão para testar uma possível colisão entre objetos presentes em nós fisicamente distantes.

Essa filtragem gera uma redução significativa na quantidade de testes efetuados por iteração e o ganho computacional frente ao método de força bruta fica mais evidente conforme a quantidade de objetos situados no espaço aumenta.

4. Algoritmo de referência

O sistema utilizado para realizar os testes, trata-se de uma adaptação de um algoritmo já existente. Este algoritmo, criado por Bill Jacobs e publicado em sua *homepage*, com propósitos educacionais, foi desenvolvido na linguagem de programação C++ com a utilização da Interface de Programação de Aplicativos, do inglês *Application Programming Interface* (API), OpenGL.

O algoritmo em si é parte de uma lista de tutoriais desenvolvidos com a finalidade de ensinar OpenGL utilizando uma abordagem focada à programação e seu código-fonte possui licença livre, sem qualquer limitação para uso, cópia, modificação, publicação, distribuição e/ou venda. Jacobs, entretanto, deixa claro que se isenta de toda e qualquer responsabilidade relacionada ao uso do seu algoritmo.

5. Sistema Desenvolvido

O objetivo deste trabalho é tornar evidente a diferença de eficiência computacional ao realizar detecção de colisão entre objetos situados no espaço bidimensional e tridimensional utilizando-se de técnicas que utilizam estruturas de dados hierárquicas e técnicas convencionais.

O código-fonte da *octree* desenvolvida por Jacobs foi utilizado como base para o desenvolvimento dos ambientes utilizados para testes no espaço bidimensional e tridimensional e toda a parte que envolve programação foi desenvolvida no ambiente de desenvolvimento CodeBlocks utilizando a API OpenGL, dado que ambas são alternativas gratuitas.

Foram desenvolvidos quatro ambientes para realizar os testes: um ambiente bidimensional que executa os testes de detecção de colisão utilizando a estrutura de dados *quadtree*; um ambiente tridimensional que executa os testes de detecção de colisão utilizando a estrutura de dados *octree* e suas respectivas contrapartes que utilizam o método convencional de detecção de colisão por força bruta.

6. Testes realizados

Para a realização das simulações em todos os quatro ambientes foi definido um valor fixo de iterações igual à 128. No momento em que a 128ª iteração é finalizada o programa termina sua execução e imprime na tela o tempo total utilizado para realizar detecção de colisão ao longo das 128 iterações, sendo este o tempo efetivo utilizado para a execução da detecção de colisão; o total de testes efetuados, considerando que são contabilizados apenas os testes realizados com pares de esferas; uma média de testes por iteração, que consiste na quantidade total de testes dividida pelo número de iterações; e o total de colisões detectadas.

Para as estruturas de dados hierárquicas foi definida uma profundidade máxima igual à 3 e uma quantidade máxima de esferas por nó igual à 2 como critério para subdivisão do nó em nós filhos.

Foram realizados testes com 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 4096 e 8192 esferas e foram realizados cinco testes para cada uma das quantidades de esferas, de modo a se obter uma média.

7. Resultados obtidos

O programa se comportou da maneira esperada: realizando todas as 128 iterações adequadamente e imprimindo os resultados na tela posteriormente. Estes resultados foram utilizados para criar gráficos de modo a expressar com mais clareza a comparação entre a eficiência da cada algoritmo.

Para a elaboração das representações gráficas dois critérios foram avaliados: o tempo total de duração da execução de todos os testes relacionados à detecção de colisão e a quantidade total de testes realizada ao final da execução do programa.

Na Figura 4, fica claro como a quantidade de objetos influencia o tempo final resultante da soma da execução de todos os testes. Inicialmente, até o ponto em que são testados 512 objetos, ambos os algoritmos tem um tempo de execução próximo à 0, à partir deste ponto o método convencional começa a crescer de maneira exponencial, dado que, ao testar uma quantidade de 8192 objetos o método convencional requer aproximados 450 segundos para executar todos os testes contra menos de 25 segundos necessários para a *quadtree* realizar todos os cálculos.

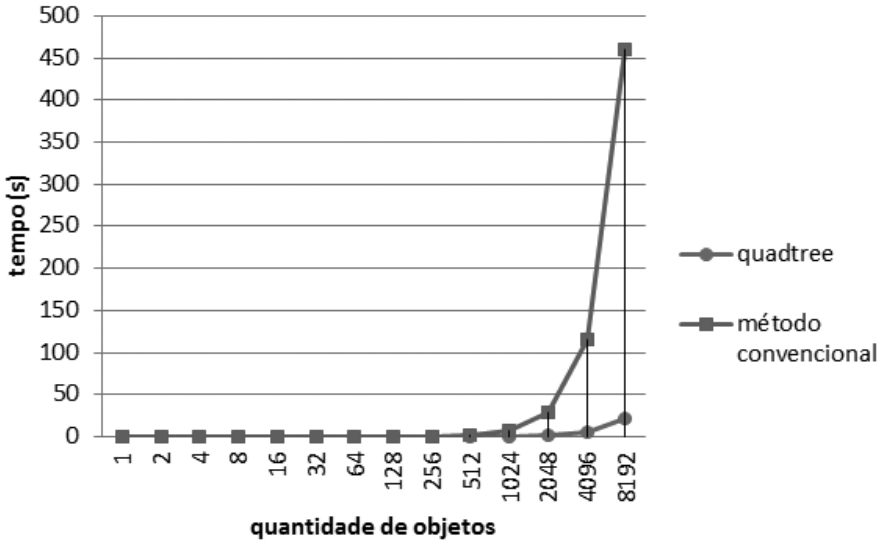


Figura 4. Q*uadtree,* relação entre tempo e quantidade de objetos

Esta diferença se dá, principalmente, pelo fato de a quantidade de testes necessárias por iteração crescer junto com a quantidade de objetos presentes no espaço.

Dada a Figura 5, que relaciona a quantidade de testes total ao final da execução do programa com a quantidade de objetos presentes no espaço, é possível perceber que o aumento de tempo presente no gráfico da Figura 4 está diretamente relacionando com a quantidade total de testes realizada no término da execução do programa. Para uma quantidade de objetos igual à 8192 a quantidade de testes contabilizados ao final da execução do programa é um valor próximo à 4,5 bilhões enquanto a *quadtree* executa cerca de 160 milhões de testes.

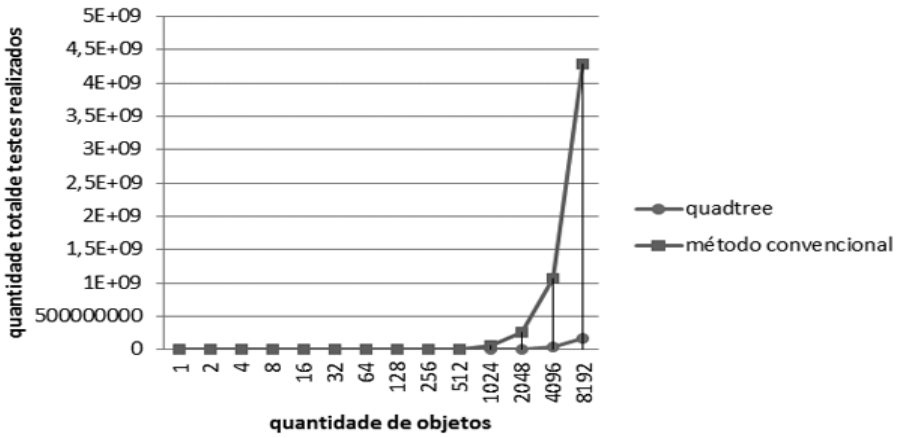


Figura 5. Q*uadtree,* relação entre tempo e quantidade de testes realizados

Haja visto uma *quatree* reduz significantemente a quantidade de testes necessários para detectar colisão, testando apenas objetos vizinhos. É visível o ganho de desempenho frente ao método de força bruta que, inicialmente, possui o mesmo desempenho que a *quadtree.*

Nos testes realizados no espaço tridimensional os resultados foram similares: ambos os métodos, nos primeiros testes, mantém um crescimento estável e desempenho semelhante, porém, quando a quantidade de objetos cresce para 512 objetos o tempo de execução total dos testes do método convencional começa a crescer exponencialmente, assim como ocorre no ambiente bidimensional.

A Figura 6 mostra a relação o tempo e a quantidade de objetos presente nas várias execuções dos ambientes tridimensionais. Assim como no sistema bidimensional foi constatado um crescimento exponencial do tempo de execução das detecções de colisão por parte do método convencional.

Foi observado, entretanto, que a *octree* obteve um desempenho maior do que a *quadtree.* Isto não se dá pelo fato de a *octree* ser mais eficiente do que uma *quatree*, mas ao fato de que o espaço para os objetos transitarem no espaço tridimensional é maior, o que resulta em uma quantidade menor de possíveis colisões, resultando assim, no notável aumento de desempenho.

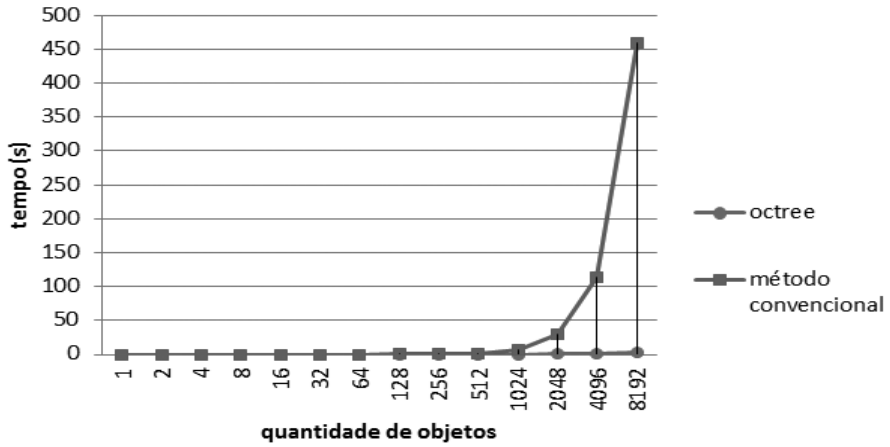


Figura 6. *Octree,* relação entre tempo e quantidade de objetos

Ao se analisar a Figura 7, assim como no caso do ambiente bidimensional, é possível realizar uma ligação entre aumento do tempo total e a quantidade de testes necessários para cada caso. Ambos crescem de maneira exponencial a partir do momento em que há mais de 512 objetos se movendo no espaço.

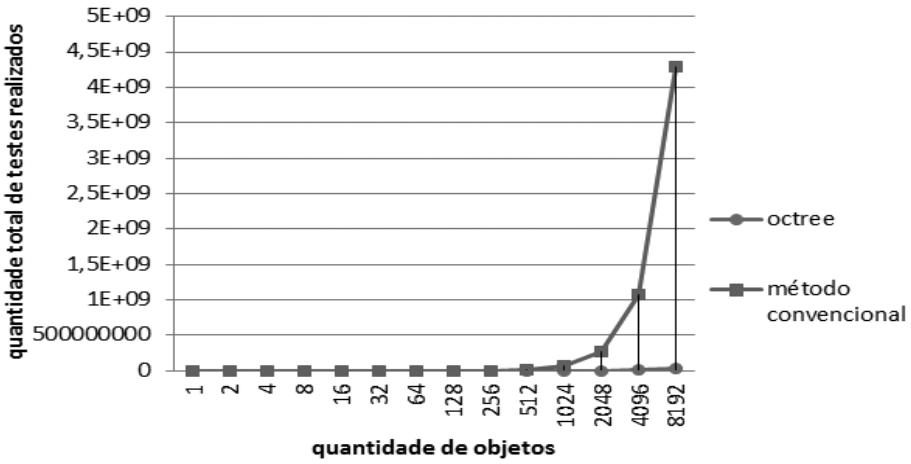


Figura 7. *Octree,* relação entre tempo e quantidade de testes realizados

8. Conclusão

Dadas diferentes situações e ambientes as estruturas de dados hierárquicas *quadtree* e *octree* se mostraram uma solução viável para detecção de colisão, frente ao método tradicional.

O método de detecção de colisão por força bruta inicialmente mostrou-se mais eficiente, até o momento em que o gerenciamento dos objetos situados no espaço começou a se tornar custoso demais devido à sua grande quantidade.

As estruturas de dados, por outro lado, com sua capacidae de indexar eventos no espaço, obtiveram uma perda mínima de desempenho. Enquanto método tradicional, à partir de determinado ponto, começou a ter um crescimento exponencial do tempo de execução para cada simulação, os ambientes que utilizaram estruturas de dados tiveram um crescimento de tempo discreto.

Visto que possuem uma natureza maleável, de modo que podem ser esculpidas para atender os mais diversos fins, estas estruturas de dados se mostraram uma solução balanceada entre robustez e eficiência para realizar e gerenciar detecção de colisão e demais eventos no espaço, tanto em ambientes bidimensionais, quanto em ambientes e tridimensionais.

Este sistema por ser refinado de modo a possibilitar a realização de detecção de colisão entre artefatos construídos por segmentos poligonais, dando ênfase à detecção de colisão entre os polígonos.

O estudo realizado ao longo deste trabalho pode ser utilizado como ponto de partida para o estudo e criação de uma técnica de tratamento de colisão ou aplicação de um *framework* responsável por tratamento de colisão já existente. A possibilidade de se realizar tratamento de colisão abre uma gama de possibilidades no campo da simulação, de modo a possibilitar diversos cenários e situaçõe como a simulação de reações físicas,

Referências

Angel, E (2000), Interactive Computer Graphics - A Top-Down Approach with OpenGL. 2. ed. Reading: Addison-Wesley.

Almeida, Vitor Pinheiro de (2011), Detecção de colisão através de Octree, http://www.puc-rio.br/pibic/relatorio_resumo2007/resumos/MAT/vitor_pinheiro_almeida.pdf

Azevedo, Eduardo; Conci, Aura (2003), Computação gráfica: teoria e prática. Rio de Janeiro: Elsevier; Campus.

Black, Paul E. (2011), Dictionary of Algorithms and Data Structures, http://xlinux.nist.gov/dads/

Drozdek, Adam (2002), Estrutura de dados e algoritmos em C++. São Paulo: Thomson.

Eder, J. (2011), Octree, http://www.cg.tuwien.ac.at/studentwork/VisFoSe98/eder/octree.htm

Foley, James D. (1995), Computer graphics: principles and practice. 2nd ed. California: Addison-Wesley, 1995.

Foley, James D. (2000), Introduction to computer graphics. New York, USA: Addison-Wesley.

Gonzalez, Rafael C.; Woods, Richard E (2000), Processamento de imagens digitais. São Paulo: Edgard Blücher.

Jacobs, Bill (2012), OpenGL Tutorial, http://www.videotutorialsrock.com/index.php

Kastelie, Willian P. (1989), The Pair tree: a parallel architecture for image representation based on symmetric recursive indexing. Canada: Simon Fraser University.

Koffman, Elliot B.; Wolfgang, Paul A. T. (2008), Objetos, abstração, estruturas de dados e projeto usando C++. Rio de Janeiro: LTC.

Marroquim, Ricardo G. (2011), Detecção de Colisão, http://www.lcg.ufrj.br/Cursos/jogos/04-deteccao-colisao.pdf

Mendes, Vilson B. (2011), Noções Básicas sobre Métodos de Modelagem, http://www.ic.uff.br/~aconci/model.htm

Milanez, Bruno Abel. (2009), Interpolação Aplicada À Animação Gerada Por Computador. Criciúma, SC: UNESC.

Novak, Jeannie (2010), Desenvolvimento de games. Tradução Pedro Cesar de Conti; revisão técnica Paulo Marcos Figueiredo de Andrare. 2nd ed. São Paulo: Cengage Learning.

Nyhoff, Larry R. (2005), ADTs, data structures, and problem solving with C++. 2nd ed Upper Saddle River, NJ: Prentice Hall.

Penton, Ron (2003), Data Structures For Game Programmers. Ohio: Premier Press.

Preiss, Bruno R. (2011), Data Structures and Algorithms with Object-Oriented Design Patterns in C++, http://www.brpreiss.com/

Quadtree (2011), In: Wikipédia: A Enciclopédia Livre, http://en.wikipedia.org/wiki/Quadtree

Sherrod, Allen (2007), Data structures and algorithms for game developers. Massachussetts: Charles River Media.

Suter, Jaap (2011), Introduction to Octrees, http://www.flipcode.com/archives/Introduction_To_Octrees.shtml

Wagner, Harley (2012), Quadtrees e Octrees, http://www.inf.ufsc.br/~visao/1998/harley/octree.htm

Xiang, Zhigang; Plastock, Roy A. (2000), Schaum's Outline of Theory and Problems of Computer Graphics. Ohio: McGraw-Hill Professional.

_____ (2011), Gamma: Geometric Algorithms for Modeling, Motion and Animation, http://gamma.cs.unc.edu/