UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

DETERMINAÇÃO DE SUPERFÍCIES VISÍVEIS PARA JOGOS NA PLATAFORMA PLAYSTATION USANDO ÁRVORES BSP

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

MARCELO ODEBRECHT

BLUMENAU, JUNHO/2001

2001/2-33

DETERMINAÇÃO DE SUPERFÍCIES VISÍVEIS PARA JOGOS NA PLATAFORMA PLAYSTATION USANDO ÁRVORES BSP

MARCELO ODEBRECHT

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Paulo César Rodacki Gomes — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Paulo César Rodacki Gomes

_____

Prof. Dalton Solano dos Reis

_____

Prof. Miguel Alexandre Wisintainer

DEDICATÓRIA

Este trabalho é dedicado aos meus pais.

AGRADECIMENTOS

Ao professor Paulo César Rodacki Gomes, meu orientador, pela motivação, dedicação e paciência.

Ao professor Miguel Alexandre Wisintainer, meu co-orientador, pela confiança, dedicação e entusiasmo.

Aos amigos e professores que contribuíram para a minha formação.

À Beatriz pelo apoio.

À comunidade de desenvolvimento para o *Playstation*, em especial Loser, Hitmen, Obiwahn e Psxdev, pelas informações.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIAÇÕES

AR – *Action Replay*;

BSP *Tree – Binary Space Partitioning tree*;

CD-ROM – *Compact Disc Read Only Memory;*

CPU – *Central Processing Unit*;

DCT – *Discrete Cosine Transform*;

DMA – *Direct Memory Access*;

GS – *Game Shark*;

GPU – *Graphics Processing Unit*;

GTE – *Geometry Transformation Engine*;

ISO – *International Organization for Standardization*;

JPEG – *Joint Photographic Expert Group*;

MDEC – *Motion Decoder*;

MPEG – *Moving Pictures Expert Group*;

OS – *Operational System*;

RGB – *Red Green Blue*;

RISC – *Reduced Instruction-Set Computing*;

SPU – *Sound Processing Unit*;

ULA – Unidade de Lógica e Aritmética.

RESUMO

O presente trabalho trata do uso de árvores *BSP* em jogos interativos. As árvores *BSP* (*Binary Space Partition Trees*) são estruturas de dados que subdividem um espaço n-dimensional em sub-espaços e são comumente usadas na determinação de superfícies visíveis para renderização em tempo real de jogos interativos. É apresentado um estudo teórico sobre árvores *BSP* e um resumo das principais características da plataforma de hardware *Playstation*, bem como das ferramentas de desenvolvimento disponíveis. Para demonstrar o uso de árvores *BSP*, é implementado um protótipo de jogo interativo bidimensional no console de videogame *Playstation*.

ABSTRACT

The present work deals with the usage of BSP trees in interactive computer games. BSP (i.e. Binary Space Partition Trees) are data structures which divide an n-dimension space in sub-spaces called half-spaces. It is commonly used for visible surface determination in real time rendering of 3D interactive video games. Along with a theoretical study on BSP trees, a review of the main characteristics of the Playstation hardware is presented, as well as a review of its main software development tools. As a demonstration of BSP trees usage, a bidimensional interactive game prototype for the Playstation hardware was implemented by the author.

1 INTRODUÇÃO

Segundo Lamothe (1999) na década de 70 já existiam jogos rodando em mainframes, em forma de texto e gráficos de baixa qualidade. Os primeiros equipamentos dedicados a jogos foram os *arcades*, ou fliperamas. Em seguida foram introduzidos no mercado os consoles de vídeo games, equipamentos também dedicados a jogos. Até há pouco tempo os *arcades* apresentavam jogos com recursos de som e imagens muito superiores aos consoles de vídeo games, mas atualmente os consoles domésticos facilmente superam essas máquinas. Dentre os vários modelos deste tipo de equipamento, o *Playstation* é a plataforma mais popular no mundo.

À medida que os computadores tornam-se mais populares e as ferramentas de desenvolvimento de jogos mais fáceis de utilizar, cresce o número de desenvolvedores de jogos. Já para o desenvolvimento de jogos para consoles de vídeo games, o número de desenvolvedores não profissionais é pouco significativo, tendo em vista a arquitetura fechada desses equipamentos. Existe, porém, a possibilidade de se utilizar equipamentos customizados e softwares desenvolvidos por programadores não ligados às empresas que desenvolvem os consoles, permitindo assim o estudo desses equipamentos, e das técnicas de programação necessárias para se implementar jogos para estas plataformas.

Com o constante avanço da tecnologia, possibilitando a criação de jogos com recursos avançados de som e imagem, os jogos tridimensionais vem conquistando a preferência de muitos jogadores. Um fator de sucesso para jogos tridimensionais além da qualidade das imagens é a velocidade com que as imagens são exibidas na tela. Segundo Foley (1990) o algoritmo *Binary Space-Partitioning Trees* (árvore *BSP*) é um método extremamente eficiente para o cálculo dos relacionamentos de visibilidade entre um grupo estático de polígonos tridimensionais vistos de um ponto arbitrário.

Para a representação de cenas tridimensionais em telas bidimensionais como nos monitores de vídeo, determina-se o ponto de vista do observador, a partir do qual se desenha uma imagem em duas dimensões. Um problema clássico em computação gráfica é determinar quais linhas ou superfícies dos objetos são visíveis. Esse processo é denominado por Foley (1990) como determinação de superfícies visíveis ou eliminação de superfícies ocultas.

Wade (1997) define as árvores *BSP* como extremamente versáteis, pois são estruturas de classificação e ordenação poderosas. Seu uso abrange desde *ray tracing* e a remoção de superfícies ocultas à modelagem sólida e planejamento de movimento robótico.

Tendo em vista estes aspectos, o presente trabalho propõe-se a investigar as principais questões relativas à construção e utilização de árvores *BSP*. É desenvolvido também, um protótipo que implemente árvores *BSP* na plataforma *Playstation* para ambientes bidimensionais, de forma a demonstrar sua utilização na determinação de superfícies visíveis em jogos interativos.

1.1 OBJETIVOS DO TRABALHO

O objetivo principal deste trabalho é desenvolver um protótipo que implemente e demonstre o uso de árvores BSP em jogos interativos bidimensionais para a plataforma *Playstation*.

Os objetivos específicos do trabalho são:

a) utilizar um microcomputador padrão *IBM PC* para o desenvolvimento de um protótipo de jogo para a plataforma *Playstation*;
b) aplicar o algoritmo árvore *BSP* para a determinação de superfície visível;
c) utilizar uma técnica de renderização de gráficos bidimensionais em tempo real;
d) criar cenários bidimensionais;
e) executar o protótipo na plataforma *Playstation*.

1.2 ESTRUTURA DO TRABALHO

O primeiro capítulo apresenta uma introdução ao trabalho desenvolvido, seus objetivos e a sua organização.

O segundo capítulo apresenta as principais características de jogos interativos.

O terceiro capítulo apresenta as definições, características e o uso da árvore *BSP*.

O quarto capítulo apresenta dados da história da plataforma *Playstation*, assim como suas principais características de hardware e seus ambientes de desenvolvimento.

No quinto capítulo, apresenta-se a especificação, implementação e funcionalidades do protótipo desenvolvido.

O sexto capítulo apresenta as conclusões, dificuldades e extensões do trabalho.

2 JOGOS INTERATIVOS

Segundo Battaiola (2001), os jogos por computador podem ser definidos como um sistema composto por três partes básicas: enredo, motor e interface interativa.

O enredo define o tema, os objetivos do jogo, o qual através de uma série de passos o usuário deve se esforçar para atingir. A definição do enredo envolve profissionais de diversas áreas do conhecimento, como pedagogos, psicólogos e outros especialistas.

A interface interativa controla a comunicação entre o motor e o usuário, reportando graficamente um novo estado do jogo. O desenvolvimento da interface envolve aspectos técnicos, cognitivos e artísticos. O aspecto técnico envolve performance, portabilidade e a complexidade dos elementos gráficos. O valor artístico de uma interface está na capacidade que ela tem de valorizar a apresentação do jogo. O aspecto cognitivo está relacionado à correta interpretação da informação gráfica pelo usuário.

O motor do jogo é o seu sistema de controle, o mecanismo que controla a reação do jogo em função de uma ação do usuário. A sua implementação envolve aspectos computacionais tais como, a escolha apropriada da linguagem de programação em função de sua facilidade de uso e portabilidade e o desenvolvimento de algoritmos específicos (Battaiola 2001).

2.1 INTERFACE GRÁFICA

A interface gráfica de um jogo é fundamental para aumentar o seu realismo e o nível de engajamento do usuário ao seu ambiente. Outros fatores devem ser considerados como a facilidade de interação, a rapidez de resposta e a incorporação ou não de trilhas sonoras, som 3D e vídeos (Battaiola 2001).

2.1.1 TÉCNICAS

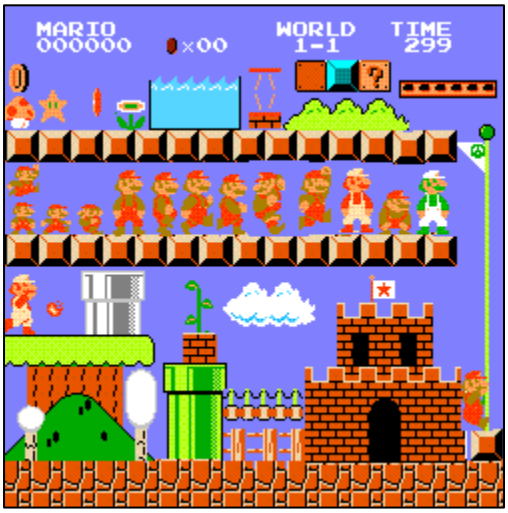
Segundo Battaiola (2001), jogos por computador tem como um dos seus principais atrativos a interatividade, o que requer a movimentação de personagens e objetos. Em computação gráfica, o termo animação se refere a qualquer alteração em uma cena em função do tempo. As principais características passíveis de alteração neste caso são: posição, forma,

cor, iluminação e material. Dentre as principais técnicas de animação, pode-se destacar a renderização de polígonos, *sprites* e quadro-a-quadro.

Na animação quadro-a-quadro, a idéia de movimento é alcançada pela exibição seqüencial de quadros a uma determinada taxa, normalmente, 24 quadros por segundo. Esta animação geralmente é pré-definida e não-interativa, portanto seu uso é limitado em jogos por computador.

O *sprite* é um objeto gráfico bidimensional que se move sobre a tela, sem deixar marcas sobre a área onde passou. Um *sprite* é composto de uma seqüência de imagens (texturas) (figura 1), geralmente retangulares, que definem a movimentação de um determinado elemento. Em função desta movimentação, as imagens são substituídas na tela.

Figura 1 – Exemplo de *Sprites* do jogo *Super Mario Bros*.



O uso de *sprites* sobrepostos ao mapa de bits de cada tela de fundo, normalmente denominado de *tile*, permite implementar uma animação rápida do objeto. Ao *sprite* são associados altura, largura, posição, estado de visibilidade, prioridade sobre outros *sprites*, transformações de escala, translação e rotação, controle de colisão com outros *sprites*, manipulação da tabela de cores, entre outras características. Os *sprites* podem ser 2D (movimentação em x e y) ou 3D (movimentação em x, y e z) (Battaiola 2001).

Em relação à interface, os jogos podem ser 2D, $2^{1/2}$D e 3D. Os jogos 2D e $2^{1/2}$D, fazem uso de *sprites* e/ou texturas e os jogos 3D renderizam em tempo real os personagens e cenários.

Jogos 2D se caracterizam essencialmente por utilizar mapas de bits, através do uso preferencial de *sprites* e de técnicas relacionadas, como *double buffering* e *scroll*. Jogos $2^{1/2}$D, ou *sprites* 3D, fazem uso extensivo de técnicas computacionalmente simples para simular uma cena 3D sem renderizações custosas. Esta técnica se divide em duas classes: *sprite*-planares e *geo-sprites*. A primeira faz uso de texturas aplicadas a objetos 3D simples, como planos. Na técnica *geo-sprites* os elementos do jogo são descritos em termos de polígonos, que são preenchidos com texturas pré definidas e armazenadas em memória.

Jogos 3D têm vários objetos constituídos de polígonos ou malhas de triângulos passíveis de renderização. Este tipo de técnica permite um aspecto visual e uma movimentação mais natural dos personagens e cenários, no entanto, é extremamente custosa em termos computacionais. Segundo Battaiola (2001), a melhor solução para contornar este problema é aumentar o poder computacional do hardware. Outra alternativa é a utilização de técnicas para melhorar o desempenho dos algoritmos de cálculos para a renderização. A determinação de superfícies visíveis é uma técnica que procura determinar dentre os objetos presentes num cenário, quais não são visíveis, por estarem ocultos por outros objetos. Este teste é realizado para evitar que áreas não visíveis do cenário sejam desenhadas ou para determinar uma ordem correta para visualização das mesmas. Segundo Shimer (1997), a árvore BSP é muito útil para a interação em tempo real com a exibição de imagens estáticas.

3 BSP

3.1 INTRODUÇÃO

A árvore *BSP* é uma árvore binária comum, utilizada para ordenação e busca de tipos geométricos, num espaço n dimensional. A árvore, na sua totalidade, representa o espaço inteiro, sendo que cada nodo contém as informações dos objetos no espaço.

3.2 CONSTRUÇÃO DE ÁRVORES BSP

Árvores *BSP* são estruturas de dados que permitem a rápida determinação de superfícies visíveis em ambientes onde o ponto de vista é alterado, enquanto os objetos permanecem estáticos. Tais ambientes podem ser cenários de jogos ou mundos virtuais, por exemplo. Assim, árvores *BSP* são muito úteis para a interação em tempo real com a exibição de imagens estáticas. Antes que as imagens sejam desenhadas, a árvore *BSP* é calculada, sendo percorrida rapidamente para a remoção de superfícies ocultas ou projeção de sombras. Com alterações ela pode ser utilizada para a manipulação de eventos dinâmicos (Shimer 1997).

A estrutura de dados árvore *BSP* torna fácil e relativamente rápida a determinação da ordem correta para a renderização. A raiz da árvore é um objeto que representa um hiperplano definido por uma face do mundo, selecionada a partir do conjunto de todas as faces que compõe o mundo. O algoritmo funciona de forma correta, não importando qual objeto (ou face) é selecionado. O objeto raiz é usado para dividir o ambiente em dois semi-espaços, um deles contém todos os objetos à frente da face na raiz da árvore, o outro contém os objetos atrás do objeto na raiz da árvore *BSP*.

Um hiperplano em um espaço n-dimensional é um objeto de dimensão n-1, que é utilizado para dividir o espaço em dois semi-espaços. Em um espaço tridimensional o hiperplano é um plano, já em um espaço bidimensional, um hiperplano é uma reta (Wade 1997).

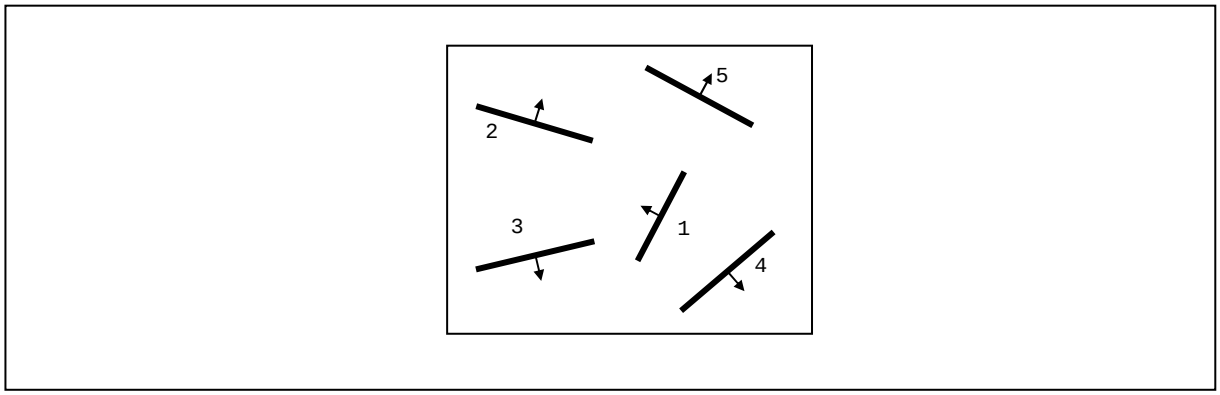
O processo de construção da árvore *BSP* divide um semi-espaço com um hiperplano qualquer, que intercepta o interior do semi-espaço, tendo como resultado dois novos semi-espaços que poderão ser novamente divididos recursivamente (Wade 1997).

Abaixo, segue um exemplo de algoritmo para a construção de uma árvore *BSP* genérica:

a) seleciona-se um hiperplano de divisão;
b) divide-se o conjunto de objetos com o hiperplano;
c) repete-se recursivamente o processo com cada novo conjunto de objetos.

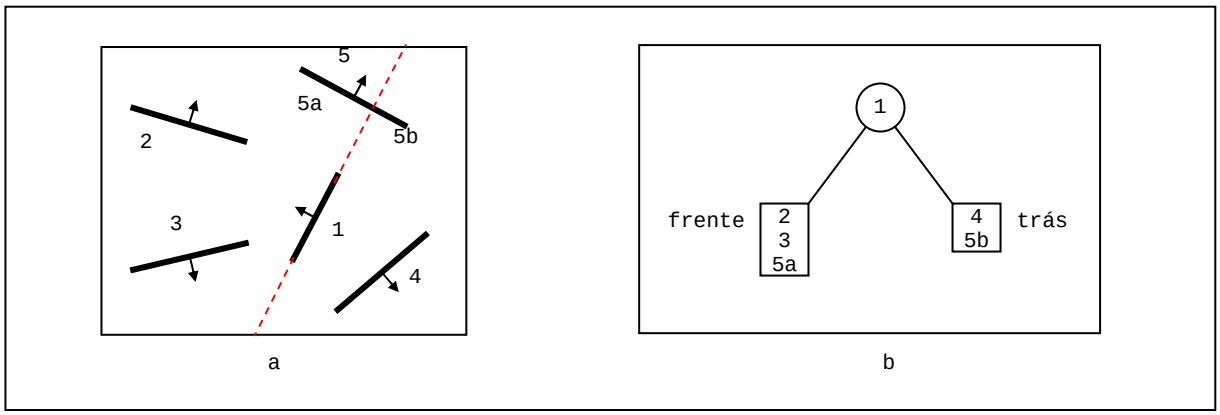
A figura 2 mostra um exemplo de um cenário bidimensional, definido pelas arestas 1, 2, 3, 4 e 5. As setas em cada aresta representam seus vetores normais, que definem qual é o lado à frente da aresta, e qual é o lado atrás da aresta. O primeiro passo para a construção de uma árvore *BSP*, é selecionar um dos objetos no espaço, neste caso, uma face, que representará a raiz da árvore.

Figura 2 – Cenário 2D.



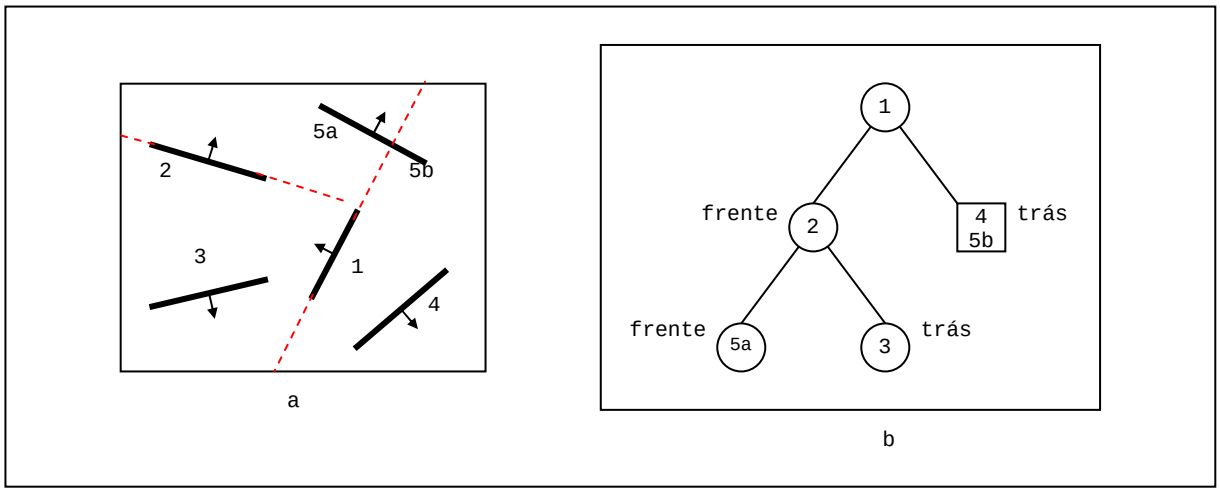
Na figura 3a, o espaço foi dividido pela aresta 1. Note-se que parte da aresta 5 está posicionada no semi-espaço à frente da aresta 1, e a outra parte está atrás. Neste caso, a aresta 5 é excluída da lista de arestas, e duas novas arestas são incluídas, 5a e 5b. À frente da aresta 1, estão as arestas 2, 3 e 5a, e atrás, estão as arestas 4 e 5b. A figura 3b mostra a árvore *BSP* correspondente ao espaço particionado pela aresta 1.

Figura 3 – a) Aresta 1 escolhida como raíz, b) Árvore *BSP*.



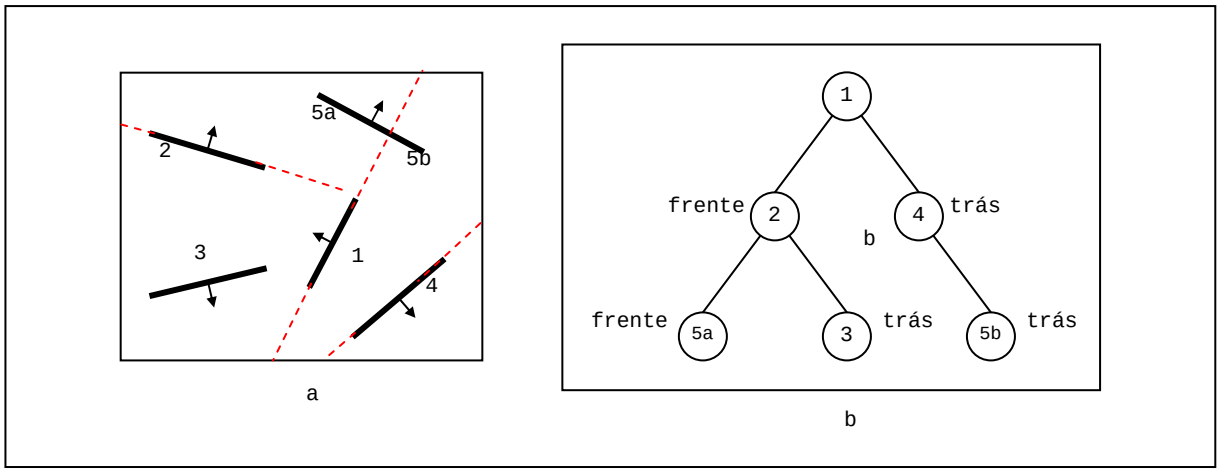
Recursivamente, os semi-espaços restantes são particionados. A figura 4a mostra o espaço após a escolha de um novo objeto para a divisão do semi-espaço à frente da aresta 1, a aresta 2, sendo que a aresta 5a posiciona-se à frente de 2 e a aresta 3 posiciona-se atrás de 2. Como não restam mais objetos a serem classificados na sub-árvore esquerda de 1, serão classificados os objetos da sub-árvore direita de 1 (figura 4b).

Figura 4 – a) Aresta 2 escolhida para dividir o espaço, b) Sub-árvore esquerda construída



A figura 5a mostra a classificação dos objetos da sub-árvore direita de 1, onde foi escolhida a aresta 4 para divisão do semi-espaço atrás da aresta 1 (sub-árvore direita de 1). A aresta 5b está posicionada atrás da aresta 4, sendo assim relacionada com a sub-árvore direita de 4. Como não existem objetos à frente de 4, e as sub-árvores direita e esquerda de 4 estão vazias, a construção da sub-árvore direita de 1 está finalizada (figura 5b).

Figura 5 – a) Escolha da aresta 4 para a divisão do espaço, b) Árvore construída.



A escolha do hiperplano de divisão depende de como a árvore será utilizada, e o grau de eficácia que se deseja obter. Normalmente escolhe-se o hiperplano a partir do conjunto de objetos, na ordem em que estes foram criados ou armazenados na memória, este método é chamado de auto divisão (Wade 1997). Para uma boa performance do algoritmo, é desejável ter uma árvore balanceada, onde cada nodo contenha aproximadamente a mesma quantidade de objetos. Para alcançar este resultado, existe um custo, pois se um objeto for interceptado pelo hiperplano, será dividido em dois novos objetos. Uma má escolha do hiperplano de divisão poderá resultar em muitas divisões, gerando muitos objetos.

Como foi visto no exemplo acima, a divisão de um conjunto de objetos com um hiperplano, é feita classificando-se cada objeto em relação ao hiperplano. Se um objeto estiver posicionado completamente de algum lado do hiperplano, então ele é inserido ao conjunto de objetos referente ao lado que ele estiver. Se o hiperplano interceptar o objeto, ele é dividido em dois novos objetos que serão incluídos no conjunto a que pertencerem. Outra possibilidade é o hiperplano ser coincidente ao objeto, neste caso, o objeto é inserido a uma lista de objetos coincidentes ao hiperplano. A decisão de quando parar a construção da árvore depende da utilização da mesma. Como critérios de parada pode-se determinar o número máximo de nodos, a profundidade da árvore, ou a totalidade de objetos.

Note-se que árvores *BSP* estão diretamente relacionadas às *Quadtrees* e às *Octrees*, que são árvores de divisão espacial que recursivamente dividem subespaços em quatro ou oito novos subespaços, respectivamente.

O quadro 1 mostra o pseudo código, em sintaxe da linguagem C, para a construção de uma árvore *BSP*, neste caso num espaço bidimensional. Ao se classificar uma face em relação à face que dividiu o espaço (hiperplano), nota-se que o resultado, pode ser COLINEAR, neste caso as duas faces são colineares, ou seja, possuem a mesma equação da reta, e a face é incluída numa lista de faces colineares.

Quadro 1 - Pseudo código para a construção de uma árvore *BSP*.

```
ArvoreBSP* ConstroiArvoreBSP(ListaFaces ListaFaces)
{
        NovoNodo = AlocarNodoDaArvore();
        FaceEscolhida = SelecionarFace(ListaFaces);
        enquanto (ListaFaces não for nula) faça
        {
                FaceAtual = SelecionarFace(ListaFaces);
                Resultado = Classificar(FaceEscolhida, FaceAtual);
                caso (Resultado)
                {
                        FRENTE: Insere(FaceAtual, ListaFrente);
                        ATRAS: Insere(FaceAtual, ListaAtras);
                        INTERCEPTA:
                        {
                                DividirFace(FaceAtual, PontoDeInterseção);
                                InserirPonto(PontoDeInterseção);
                                InserirFace(PontoInicial, PontoDeInterseção, ListaFaces);
                                InserirFace(PontoDeInterseção, PontoFinal, ListaFaces);

                        }
                        COLINEAR: Insere(FaceAtual, NovoNodo->ListaColineares);
                }
        }
        se (ListaFrente não estiver vazia) então
                NovoNodo->Frente = ConstroiArvoreBSP(ListaFrente);
        se (ListaAtras não estiver vazia) então
                NovoNodo->Atras = ConstroiArvoreBsp(ListaAtras);
        retorna NovoNodo;
}
```

No algoritmo acima, são utilizadas, além da estrutura da árvore *BSP*, duas outras estruturas. Uma lista de pontos e uma lista de faces, ambas são listas circulares duplamente encadeadas, alocadas na memória dinamicamente. No quadro 2, o pseudo código mostra o algoritmo para a criação de um novo elemento em uma lista genérica. Note-se que este algoritmo pode ser utilizado, tanto para a criação da lista de faces, quanto para a criação da lista de pontos.

Quadro 2 – Algoritmo para a criação de um novo elemento na lista circular.

```
Lista AdicionaNoFim(Lista* Cabeça)
{
        NovoElemento = AlocarMemoria();
        se (Cabeça for nula) então
        {
                NovoElemento->próximo = NovoElemento;
                NovoElemento->anterior = NovoElemento;
                Cabeça = NovoElemento;
        }
        senão
        {
                NovoElemento->proximo = Cabeça;
                NovoElemento->anterior = Cabeça->anterior;
                Cabeca->anterior->próximo = NovoElemento;
                Cabeça->anterior = NovoElemento;
        }
        retorna NovoElemento;
}
```

No algoritmo apresentado no quadro 3, é utilizada a função *SelecionaFace()*. Esta função demonstrada no quadro 3, é utilizada, para retirar uma face da lista de faces passada como parâmetro. Utilizou-se a técnica da auto divisão, onde é escolhida a face pela ordem que foi inserida na lista, retornando a primeira face, ou seja a cabeça da lista.

Quadro 3 – Pseudo código para a rotina de auto seleção de uma face

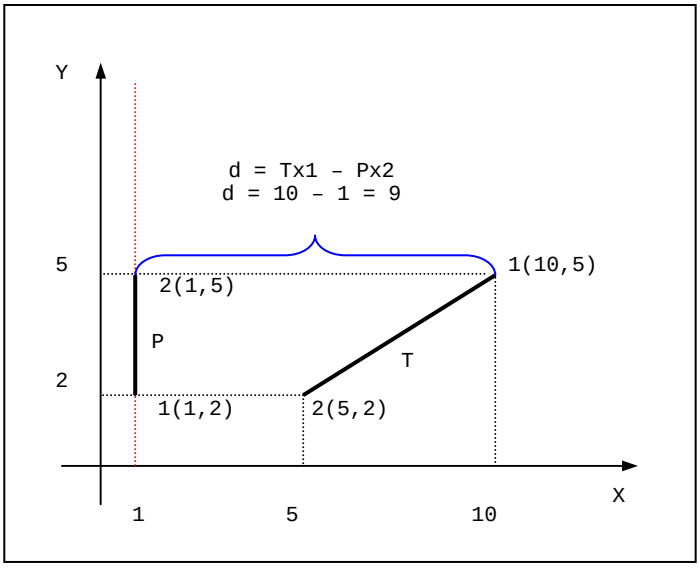
```
Lista SelecionaFace(ListaFaces Cabeça)
{
        se (Cabeça não for nula) então
                Resultado = RetiraElemento(Cabeça, *Cabeça);
        retorna Resultado;
}

ListaFaces RetiraElemento(ListaFaces Cabeça, ListaFaces Elemento)
{
        se (Cabeça igual a Elemento) então
                Cabeça = Elemento->próximo;
        senão
                Cabeça = nulo;
        Elemento->próximo->anterior = Elemento->anterior;
        Elemento->anterior->próximo = Elemento->próximo;
}
```

No algoritmo mostrado no quadro 4, é testado se a face **escolhida** é paralela ao eixo X ou ao eixo Y. Este teste é realizado, pois sendo um dos casos verdadeiro, para calcular a distância entre os pontos da face **atual** e a face **escolhida**, basta subtrair o valor X de cada ponto da face **escolhida** pelo valor X de cada ponto da face **atual**, se a face **escolhida** for paralela ao eixo X. Se o a face **escolhida** for paralela ao eixo Y, subtrai-se o valor Y de cada ponto da face **escolhida** pelo valor Y de cada ponto da face **atual**, como mostra a figura 6, onde a aresta P representa a face **escolhida**, e a aresta T representa a face **atual**.

Quadro 4 - Algoritmo para a classificação de uma face.

```
inteiro Classificar(Face Escolhida, Face Atual)
{
        se (Escolhida é paralela a X ou a Y) então
        {
                //neste caso basta subtrair X ou Y de Escolhida e Atual
                //sendo Escolhida paralela a X, subtrair X
                //caso contrário subtrair Y
                CalcularDistâncias(Escolhida, Atual);
                se (Distâncias entre Atual e Escolhida tiverem sinais opostos)
                        retorna INTERCEPTA;
        }
        senão
        {
                //neste caso é necessário calcular o coeficiente angular (a)de Escolhida
                //e o seu ponto de interseção no eixo y (b)
                //as distâncias são então calculadas  entre os pontos de Atual
                //e a equação da reta (y = ax + b) de Escolhida
                CalcularDistâncias(Escolhida, Atual);
                se (Distâncias entre Atual e Escolhida tiverem sinais opostos)
                        retorna INTERCEPTA;
        }
        CalcularDistânciaNomal(Escolhida);
        se (Distância e DistânciaNormal tiverem sinais opostos) então
                retorna ATRÁS;
        senão
                retorna FRENTE;
}
```

Figura 6 – Cálculo da distância de um ponto a uma reta paralela ao eixo Y.



Nos casos em que a face **escolhida** não for paralela a nenhum dos eixos, a distância é calculada da seguinte forma:

a) calcula-se o coeficiente angular a, da face **escolhida**;

b) calcula-se o ponto de interseção no eixo y, b, da face **escolhida**;

c) aplica-se a fórmula da distância de um ponto à reta (Lehmann 1987) (quadro 5), para cada um dos pontos da face **atual**.

Quadro 5 – Fórmula da distância de um ponto à reta.

Distância de um ponto à reta: $d = \frac{Ax + By + C}{\sqrt{A^2 + B^2}}$

onde a equação da reta: $Ax + By + C = 0$ ou $y = ax + b$

conclui-se então: $y = \frac{Ax + C}{B}$

onde: A = a; B = 1 e C = b

tem-se: $y = \frac{ax + b}{1} = ax + b$ ou $ax - y + b = 0$

$$d = \frac{ax - y + b}{\sqrt{a^2 + 1}}$$

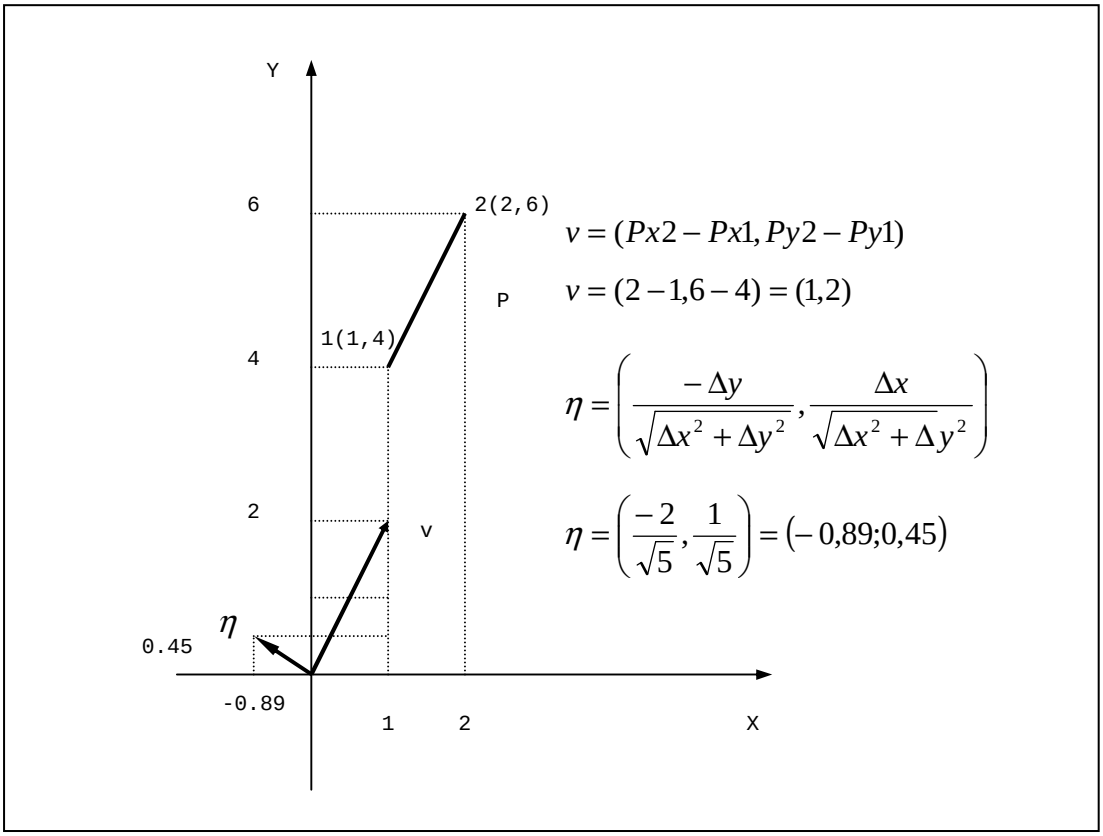
onde: $a = \frac{\Delta y}{\Delta x} = \frac{y2 - y1}{x2 - x1} = \frac{y - y1}{x - x1}$

$b = y1 - ax1$

A figura 7, mostra como é definido o vetor que representa um segmento de reta (v) e o seu vetor normal (η). O cálculo do vetor normal se faz necessário quando se deseja classificar em relação à posição no espaço, se um ponto está à frente ou atrás de uma reta. Para isso calcula-se o comprimento do vetor normal (η) ao segmento de reta com origem na origem do

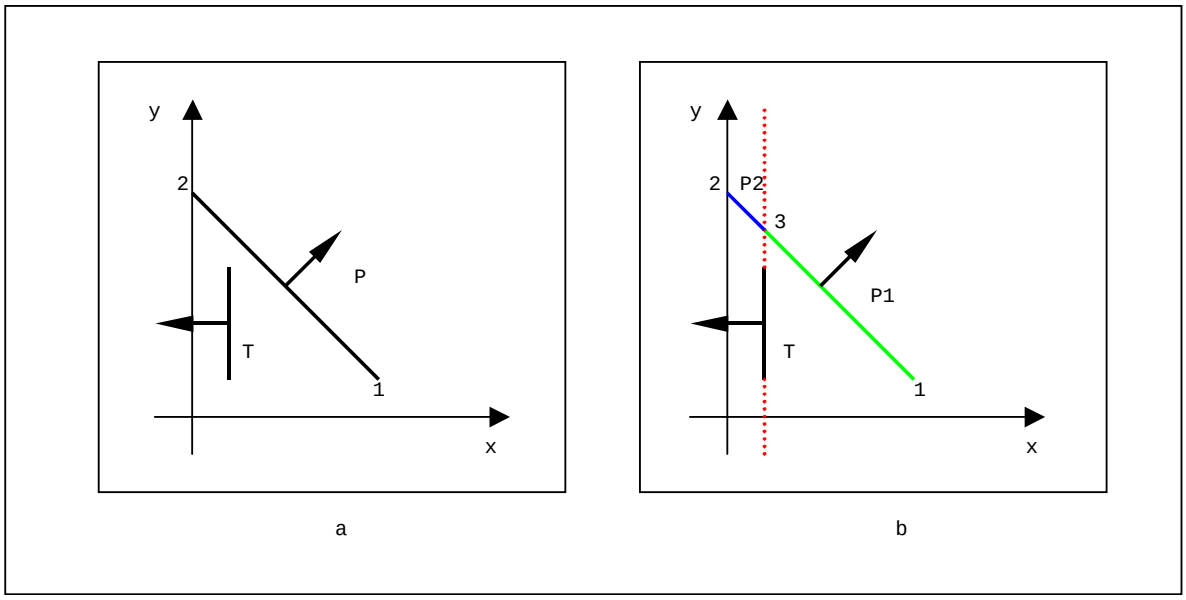
sistema de coordenadas. Depois, para cada ponto a ser avaliado, calcula-se um vetor normal a reta, passando pelo ponto considerado. É calculado o comprimento deste vetor, se seu sinal for igual ao sinal do comprimento do vetor normal original (η), então o ponto está do mesmo lado apontado pelo vetor normal da reta, ou seja está à frente da reta. Se os sinais forem opostos, o ponto está posicionado atrás da reta em questão.

Figura 7 – Cálculo do vetor e vetor normal da reta.



Para avaliar se um dado segmento de reta é interceptado por uma reta divisora do espaço, testa-se os dois pontos que formam o segmento de reta. Se os comprimentos dos dois vetores normais em relação à reta divisora, formados pelos dois pontos de definição do segmento testado, tiverem sinais opostos, este segmento é interceptado pela reta. Nesse caso é necessário dividir o segmento de reta em dois novos segmentos (figura 8).

Figura 8 – Divisão do segmento de reta P em P1 e P2.



O cálculo da interseção de duas retas é necessário em casos semelhantes ao mostrado na figura 8a. Após a divisão do espaço pela aresta T (reta divisora), verifica-se que uma parte da aresta P está posicionada à frente da aresta T. A outra parte está posicionada atrás da aresta T. Como foi visto anteriormente, quando ocorre a interseção entre a aresta que divide o espaço e a aresta escolhida, é necessário dividir a aresta escolhida em duas novas arestas. Na figura 8b as duas arestas formadas pela divisão da aresta P estão representadas pelas arestas P1 e P2. O ponto inicial da aresta P1 é o ponto 1, e o seu ponto final é o ponto de interseção das arestas P e T, o ponto 3. O ponto inicial da aresta P2 é o ponto 3 e o ponto 2, o seu ponto final. O quadro 6 mostra o pseudocódigo para o cálculo do ponto de interseção entre dois segmentos de reta.

Quadro 6 – Cálculo do ponto de interseção entre duas retas

```
Inter(Face P, Face T, real& x, real& y)
{
        aP = (Py2 – Py1) / (Px2 – Px1);
        bP = -(aP * Px1) + Py1
        aT = (Ty2 – Ty1) / (Tx2 – Tx1);
        bT = -(aT * Tx1) + Ty1
        se (bP igual a bT)
        e  (aP igual a aT) então
                retorna COLINEARES;
        se (aP igual a aT)
                retorna PARALELAS;
        x = ((Ty1 – Py1) + (aP * Px1) – (aT * Tx1)) / (aP - aT);
        y = ((aP * aT * Px1) – (aT * Py1) – (aP * aT * Tx1) + (aP * Ty1)) / (aP- aT);
        retorna INTER;
}
```

Com a finalidade de melhorar o desempenho do algoritmo e evitar divisões por zero no cálculo da interseção, é necessário realizar alguns testes. Se ΔX da reta P for igual a zero, por exemplo, no cálculo do coeficiente angular desta reta haverá um divisão por zero, como esta reta é paralela ao eixo Y, não há necessidade de calcular o valor de X do ponto de interseção, pois este será Px1 = Px2 = x. No início do algoritmo pode-se ainda testar se P e T são colineares, apenas verificando se os valores de X, quando paralelas ao eixo Y, ou os valores de Y, quando paralelas ao eixo X, de ambas as retas são iguais. O quadro 7 mostra as fórmulas utilizadas para o cálculo do ponto de interseção de duas retas P e T.

Quadro 7 – Fórmulas para o cálculo do ponto de interseção.

Vimos que $aP = \frac{y - yP1}{x - xP1}$ isolando-se o y: $y = ap(x - xP1) + yP1$ e

$aT = \frac{y - yT1}{x - xT1} \therefore y = aT(x - xT1) + yT1$ então $aP(x - xP1) + yP1 = aT(x - xT1) + yT1$

Resolvendo: $x = \frac{(aPxP1 - aTxT1) + (yT1 - yP1)}{aP - aT}$

Substitui-se o x em: $y = aT(x - xT1) + yT1$

$$y = at\left(\frac{(aPxP1 - aTxT1) + (yT1 - yP1)}{aP - aT} - xT1\right) + yT1$$

Tem-se: $y = \frac{aPaTxP1 - aTyP1 - aPaTxT1 + aPyT1}{aP - aT}$

3.3 PERCORRIMENTO DA ÁRVORE

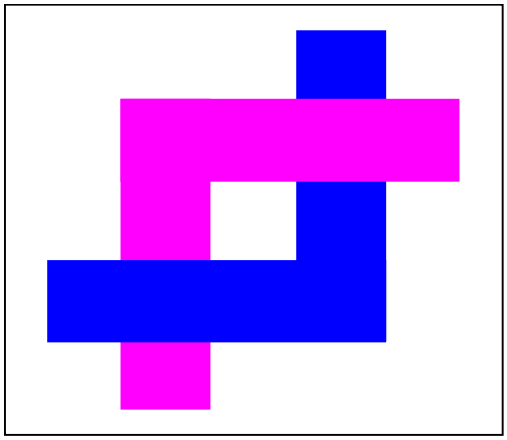
A maior vantagem das árvores *BSP* segundo (Chin 1995), é o percorrimento da árvore na ordem de trás para frente, a partir de um ponto de vista arbitrário. Para desenhar o conteúdo da árvore, pode-se realizar o percorrimento na ordem de trás para frente, como no algoritmo do pintor, ou na ordem frente para trás, para o uso com o algoritmo *scanline*. O pseudo código no quadro 8, mostra um exemplo de algoritmo para o percorrimento da árvore *BSP* na ordem de trás para a frente.

Quadro 8 – Algoritmo para o percorrimento da árvore *BSP*.

```
void PercorreArvoreBSP(ArvoreBSP* Arvore, real x, real y)
{
        Resultado = Classificar(x, y, SelecionaFace(Arvore->ListaFaces));
        se (Resultado igual a FRENTE) então
        {
                //o ponto de vista está a frente da face
                PercorreArvoreBSP(Arvore->tras, x, y);
                DesenhaArvoreBSP(Arvore->ListaFaces);
                PercorreArvoreBSP(Arvore-frente, x, y);
        }
        senão
        {
                //o ponto de vista está atrás da face, ou coincide com ela
                PercorreArvoreBSP(Arvore-frente, x, y);
                DesenhaArvoreBSP(Arvore->ListaFaces);
                PercorreArvoreBSP(Arvore->tras, x, y);
        }
}
```

A renderização na ordem de trás para frente, desenha primeiro os objetos mais distantes do ponto de vista do observador, seguido pelos que estiverem mais próximos. Desta forma, representa uma aproximação do algoritmo do pintor (Foley 1990). Ao desenhar numa tela, o pintor desenha primeiro o segundo plano, para então desenhar o primeiro plano. A condição para este método funcionar corretamente, é que dois objetos visíveis quaisquer possam ser separados por um hiperplano. A figura 9, mostra um exemplo onde a separação não é possível. Neste caso o algoritmo do pintor não irá desenhar os objetos corretamente. A construção da árvore *BSP* dividirá os objetos sobrepostos em objetos separados. Os objetos representados pelos nodos da árvore serão corretamente ordenados para serem desenhados na ordem de trás para frente (Eberly 2001).

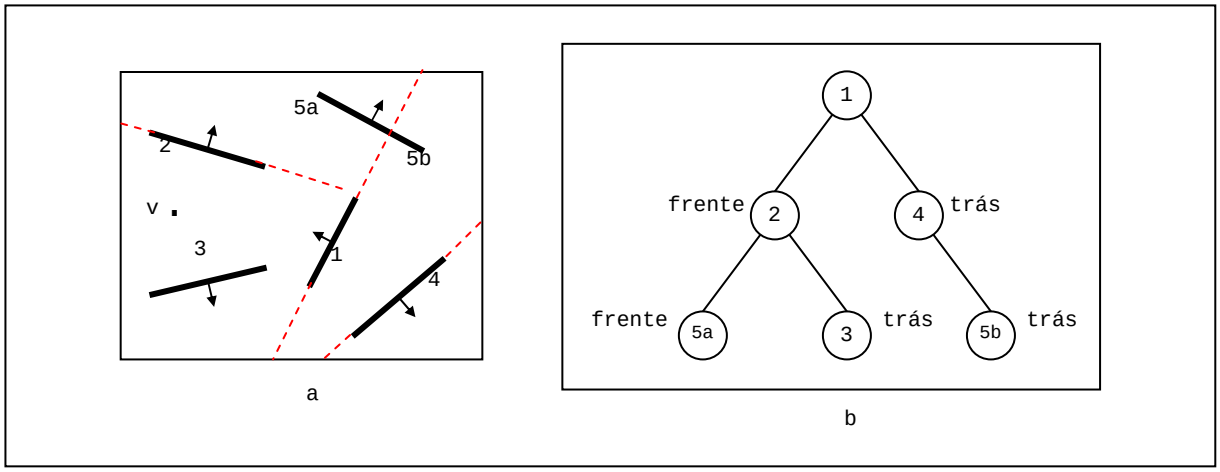
Figura 9 – Situação não suportada pelo algoritmo do pintor.



Fonte: Eberly (2001)

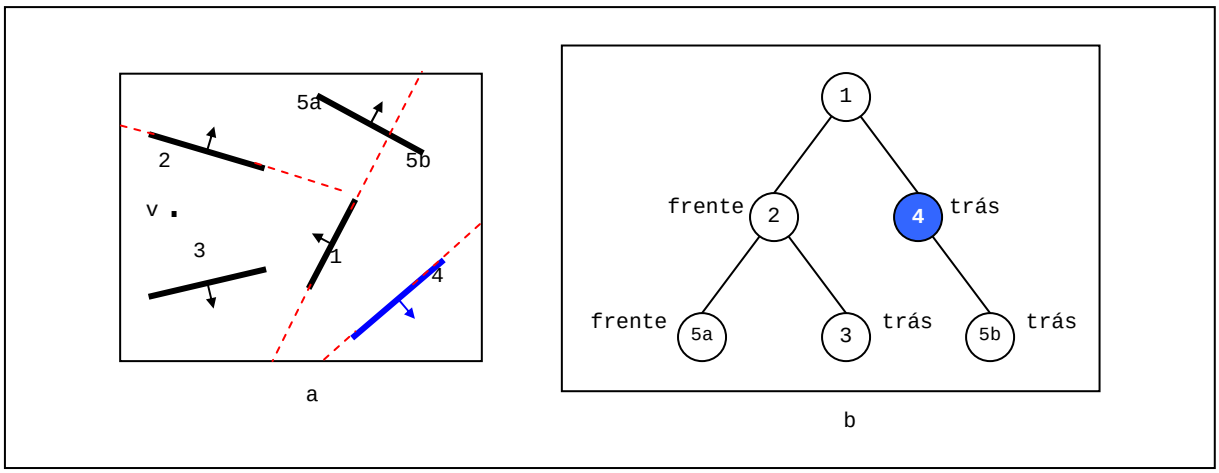
Na figura 10, é mostrado o primeiro passo para o percorrimento da árvore *BSP*, onde o ponto de vista é representado por **v** (figura 10a). Após testar-se a posição de **v** em relação ao nodo raiz da árvore, que representa a face 1, verifica-se que este está posicionado à frente da face 1. O algoritmo, executado recursivamente, percorre então a sub-árvore atrás do nodo raiz, sendo o nodo 4 o próximo nodo percorrido.

Figura 10 – a) Cenário a ser desenhado b) Árvore *BSP* representando o cenário.

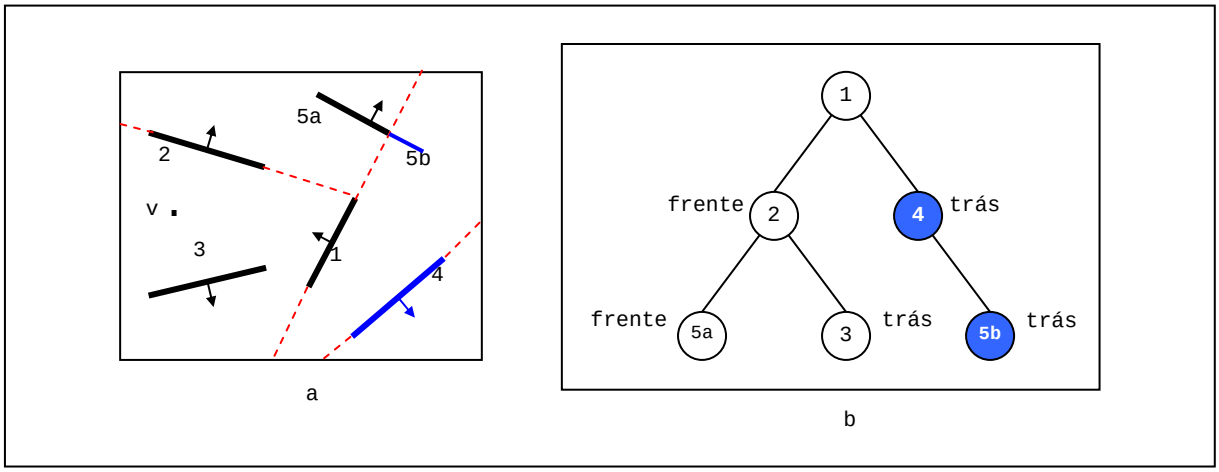


Verifica-se no passo seguinte que o ponto de vista **v**, está posicionado atrás da face 4, assim o percorrimento continuaria a partir da sub árvore à frente do nodo 4 (figura 11b), que neste caso não existe, então a face 4 é desenhada (figura 11a).

Figura 11 – a) Face 4 desenhada b) Percorrimento da árvore *BSP*, nodo 4.

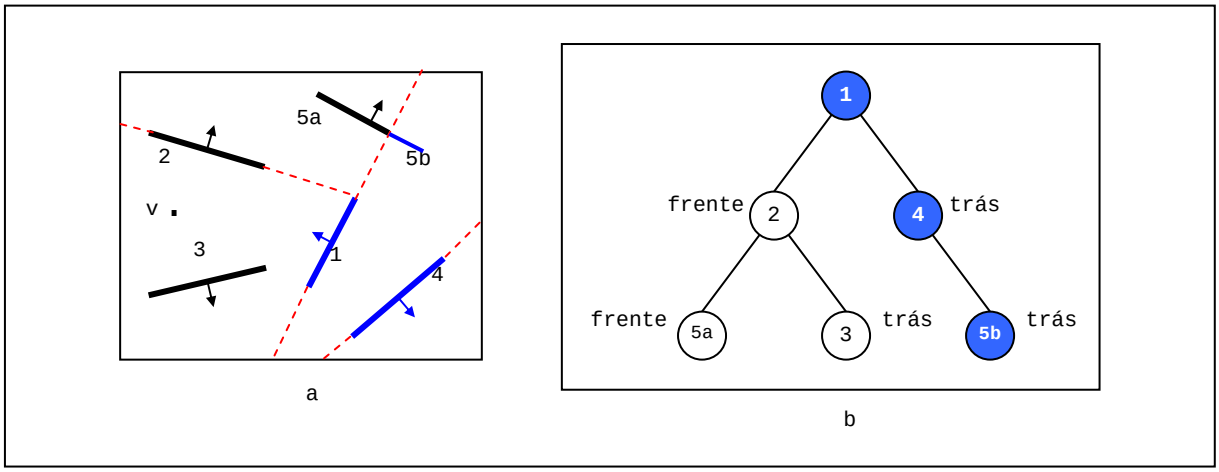


O próximo nodo percorrido é o nodo 5b (figura 12b), como este nodo não possui filhos, a face 5b é desenhada (figura 12a).



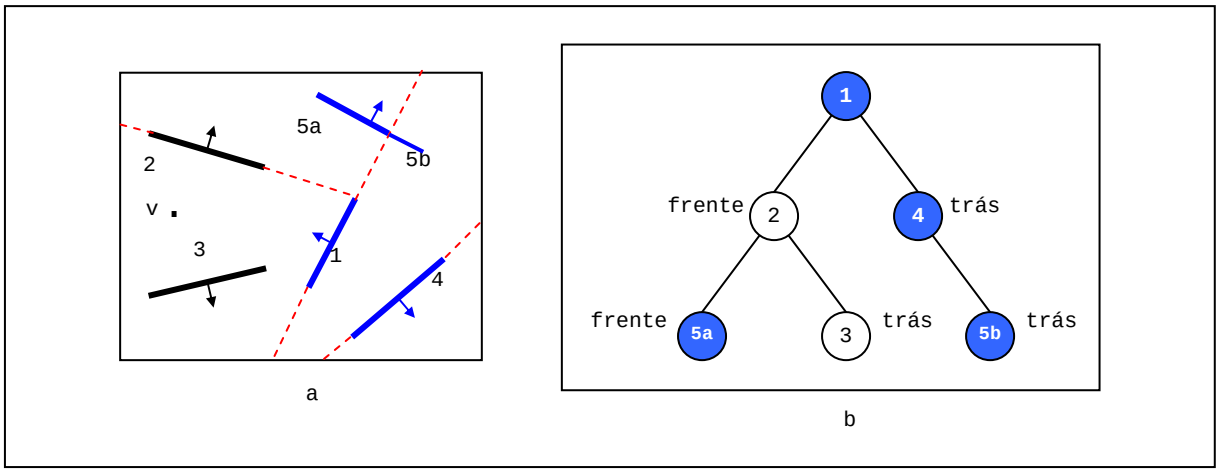
Ao retornar ao nodo 1 (figura 13b), a face 1 é desenhada (figura 13a). Em seguida, é percorrida a sub árvore à esquerda do nodo 1, que representa o semi-espaço à frente da face 1. Desta forma, o nodo 2 é o próximo a ser percorrido.

Figura 13 – a) Face 1 desenhada b) Percorrimento da árvore *BSP*, nodo 1.



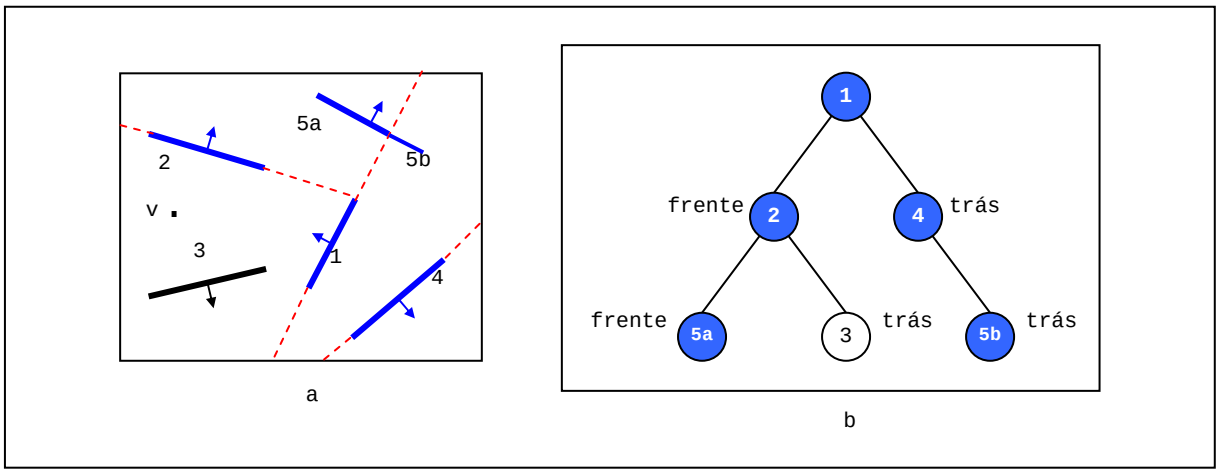
Verifica-se que o ponto de vista **v** está posicionado atrás da face 2. Assim, a sub-árvore à esquerda deste nodo (semi-espaço à frente da face 2) é percorrida. Já que o nodo 5a não possui mais nodos filhos (figura 14b), a face 5a é desenhada (figura 14a).

Figura 14 – a) Face 5a desenhada b) Percorrimento da árvore *BSP*, nodo 5a.



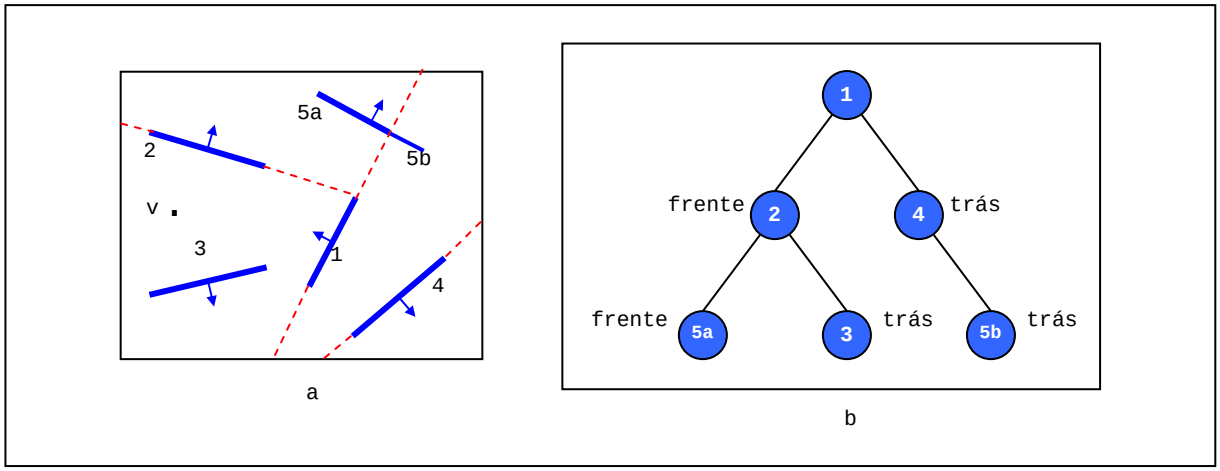
No passo seguinte, a face 2 é desenhada (figura 15a), e o nodo 3 é o próximo a ser percorrido (figura 15b).

Figura 15 – a) Face 2 desenhada b) Percorrimento da árvore *BSP*, nodo 2.



Após testar a posição do ponto de vista **v** em relação à face 3, verifica-se que o nodo 3 não possui filhos (figura 16b). Assim, a face 3 é desenhada (figura 16a), concluindo o percorrimento da árvore *BSP*.

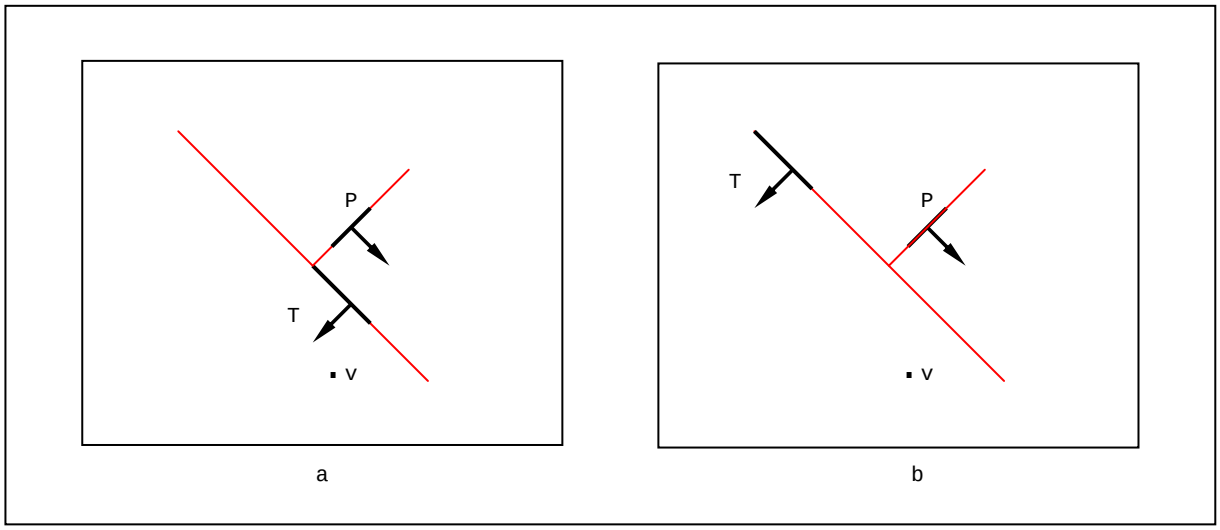
Figura 16 – a) Face 3 desenhada b) Percorrimento da árvore *BSP*, nodo 3.



Concluído o percorrimento da árvore *BSP*, a ordem correta para a renderização da faces no exemplo dado é: 4, 5b, 1, 5a, 2 e 3.

Nota-se que a ordenação de trás para frente, é independente da distância do ponto de vista. As figura 17a e 17b mostram um exemplo onde o objeto P está situado completamente no semi-espaço oposto ao do ponto de vista **v**. Ao se percorrer a árvore, o objeto P será desenhado por primeiro e então o objeto T é desenhado (figura 17a). Já na figura 17b, o objeto P continua sendo desenhado antes do objeto T, apesar de estar mais distante do ponto de vista.

Figura 17 – a) Objeto T próximo ao ponto de vista **v** b) Objeto T distante ao ponto de vista **v**.



Fonte: Adaptado de Chin (1995).

Mais importante é observar que a ordenação de trás para a frente não é independente apenas quanto à distância, é também independente quanto à direção. Isto significa que dado um ponto de vista, será gerada a mesma ordem, não importando a direção do ponto de vista. Nos exemplos vistos, apenas a posição do ponto de vista **v** é necessária para o percorrimento da árvore e não a sua direção (Chin 1995).

Conclui-se que o percorrimento da árvore *BSP* retorna apenas uma ordenação, não revelando quais objetos estão dentro campo de visão. É garantido apenas a ordem correta para o desenho dos objetos do cenário, desta forma objetos distantes do ponto de vista serão ocultados pelos objetos mais próximos.

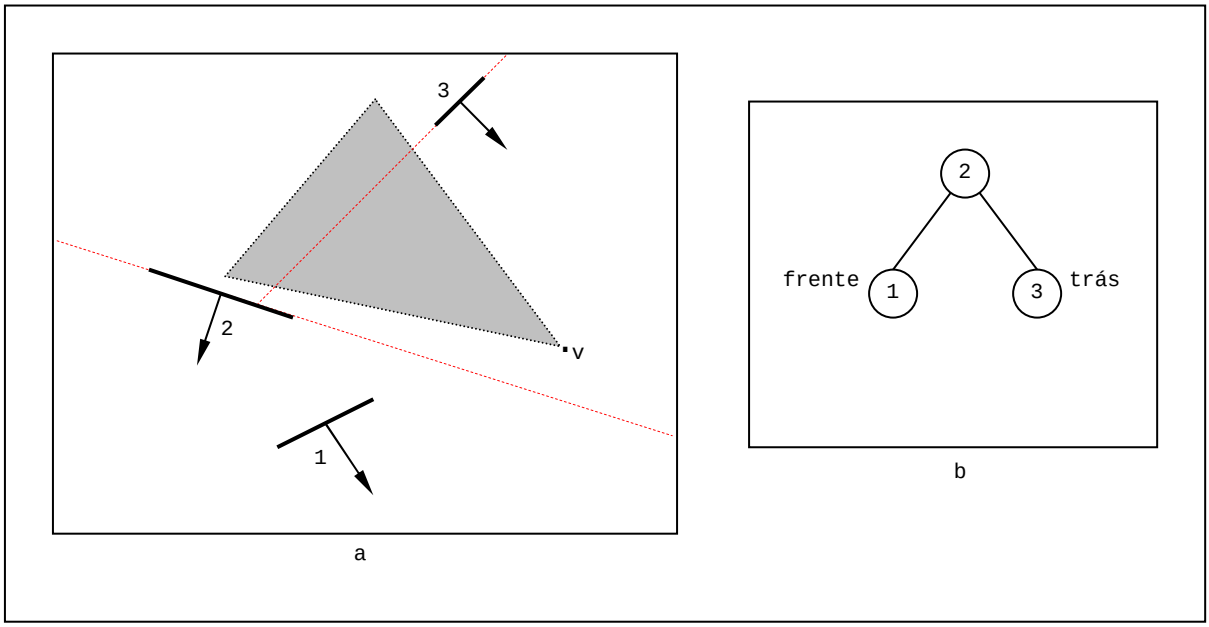
Para otimizar a renderização dos objetos da árvore *BSP*, Chin (1995) descreve os métodos *back-face culling* e *view-frustum culling*.

O método *back-face culling*, pode ser feito sem custo durante o percorrimento da árvore *BSP*, para isso testa-se a posição do ponto de vista do observador, se ela estiver no semi-espaço atrás de um objeto, este não será desenhado. Na figura 16a, vista anteriormente, o observador **v** está atrás dos objetos 2, 3, 4, 5a e 5b, que não serão desenhados. Nota-se, que mesmo realizando o método *back-face culling*, a árvore é percorrida por completo. Para auxiliar o método *back-face culling*, pode-se utilizar o método *view-frustum culling*. Este método verifica o posicionamento de todos os vértices do campo de visão do observador. Se os vértices estiverem do mesmo lado, então a sub árvore correspondente ao lado oposto é ignorada (Chin 1995).

Na figura 18a o campo de visão, definido pelo polígono cinza, está posicionado atrás do objeto 2, assim o objeto 2 não será desenhado (método *back-face culling*) e a sub-árvore correspondente à sua frente, não será percorrida. O percorrimento será feito recursivamente, a partir da sub-árvore direita do objeto 2. Após a face 3 dividir o semi-espaço atrás da face 2, nota-se que o campo de visão do observador é interceptado pela face 3, sendo assim desenhada. A ordem para a renderização resultante consiste apenas no objeto 3. Comparando-se com a lista gerada sem a utilização do método *view-frustum culling*, ou seja, 3, 2 e 1, conclui-se que o método é eficaz. É importante observar que este método não garante que os objetos posicionados fora do campo de visão sejam omitidos. Como foi visto na figura

12a, o objeto 3 está posicionado fora do campo de visão do observador v, mas foi incluído na lista de objetos ordenada (Chin 1995).

Figura 18 – a) Cenário b) Árvore *BSP* correspondente ao cenário da figura 18a.



Fonte: Adaptado de Chin (1995)

4 PLATAFORMA PLAYSTATION

Lançado há sete anos, o *Playstation* (figuras 19 e 20) é o console doméstico para *videogames* mais vendido no mundo. Mesmo após o lançamento de seu sucessor, o *Playstation* 2, novos jogos e acessórios são lançados freqüentemente. Neste capítulo serão vistas suas principais características, assim como os meios para o desenvolvimento de aplicativos e jogos para esta plataforma de *hardware*.

Figura 19 – Console *Playstation* (vista frontal).

Figura 20 – Console *Playstation* (vista traseira).

4.1 HISTÓRICO

Em 3 de dezembro de 1994 a *Sony Computer Entertainment Inc*. lançou no Japão o seu primeiro *videogame*, o *Playstation*, que depois foi lançado na América do Norte e na Europa em setembro de 1995. Até 28 de abril de 2000 haviam sido lançados 2.817 títulos de jogos no Japão, 830 na América do Norte e 860 na Europa. Até 26 de julho de 2001, as vendas do console ultrapassaram 85 milhões de unidades em todo mundo. Em 7 de julho de 2000, o console foi remodelado, tendo o seu tamanho reduzido a um terço do original, e foi chamado *PSOne* (figura 21). Em 4 de março de 2000 foi lançado no Japão o *Playstation* 2, console de 128 bits (Sony Computer Entertainment Europe 2001).

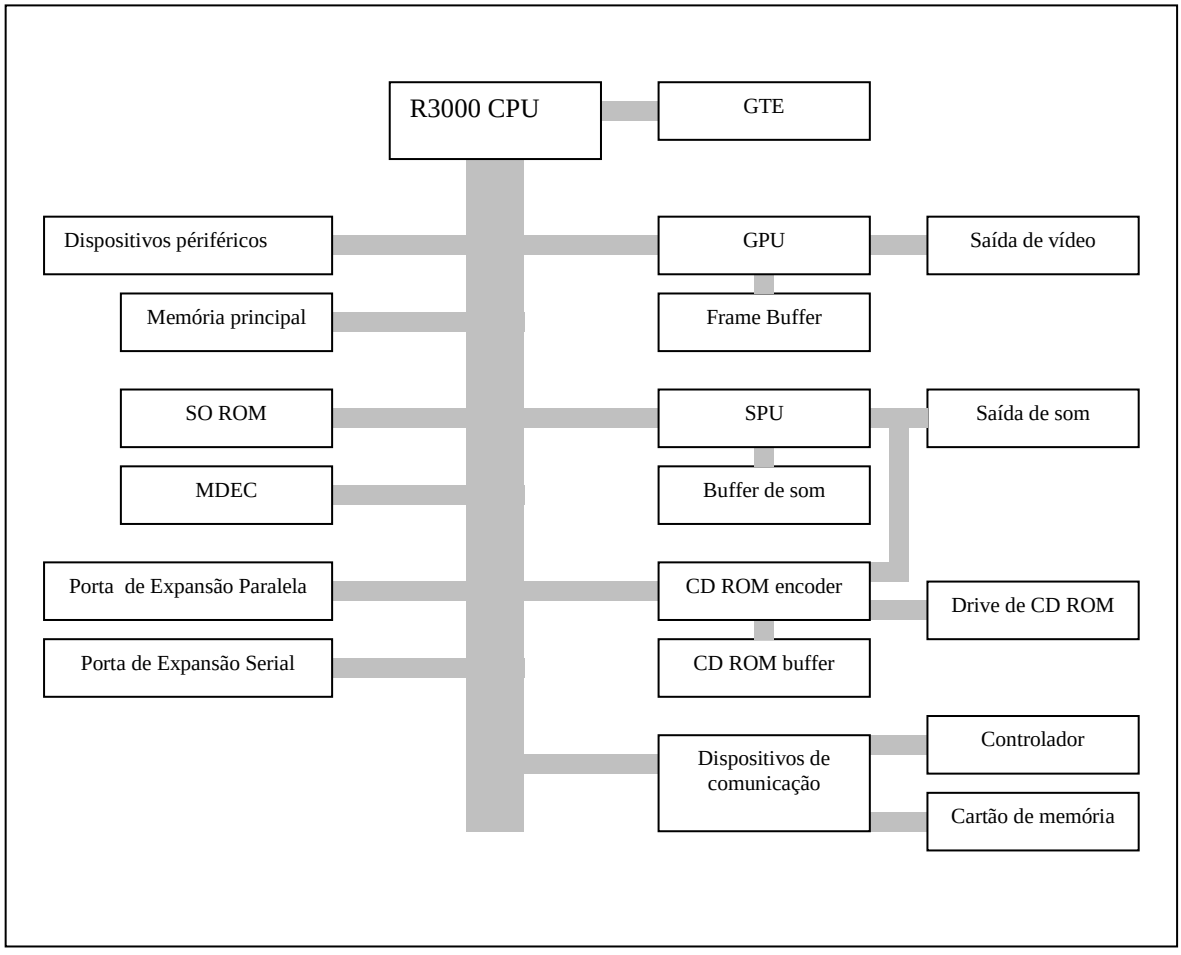
Figura 21 – *PSOne*.

4.2 HARDWARE

A *CPU* do *Playstation* (figura 22) é um processador R3000A modificado, de arquitetura *MIPS*, produzido pela *LSI Logic Corporation*. O R3000A é um processador de 32 bits de tecnologia *RISC* (*Reduced Instruction-Set Computing*), seu *clock* é de 33,8688 MHz. Sua performance de operação é de 30 milhões de instruções por segundo (30 *MIPS*). Possui um cache interno de instruções de 4 KB, um cache de dados de 1 KB e um *BUS* de

transferência de 132 MB/s. Possui ainda internamente uma unidade de lógica e aritmética (ULA) e um *shifter*. O *Playstation* possui dois coprocessadores, o *System Control Coprocessor* e a Unidade de Processamento Gráfico (GPU - *Graphics Processing Unit*). Todos os serviços para o controle da *CPU* e das unidades de hardware do *Playstation* são providos através de funções na linguagem C (Sony Computer Entertainment Europe 1998).

Figura 22 - Arquitetura do *Playstation*.



Fonte: Adaptado de Sony Computer Entertainment Inc. (1997b)

O *Playstation* possui 512 KB de memória ROM, onde estão armazenados o núcleo de seu sistema operacional (PSX OS) e o *boot loader*, cujo acesso não é permitido. O *PSX OS* é um sistema operacional multitarefa, podendo realizar múltiplo processamento de forma assíncrona. O controle multitarefa é apropriado para realizar, por exemplo, a leitura do dispositivo de CD-ROM ou a execução de música de fundo. Após a carga do sistema

operacional, a execução multitarefa estará desabilitada, podendo ser habilitada em seguida. O sistema de arquivos é acessado através de um driver de dispositivo que permite múltiplos sistemas de arquivos coexistirem, cujos arquivos estão armazenados no CD-ROM. Esse sistema de arquivos é baseado na ISO 9660 nível 1 (Sony Computer Entertainment Europe 1998).

O controlador *DMA* está anexado à *CPU*. Este controlador é responsável pela transmissão de dados entre a memória e os dispositivos, de acordo com instruções da *CPU* (Sony Computer Entertainment Inc. 1997b).

O motor de descompressão de dados (*MDEC*) é responsável pelos cálculos de conversão *DCT* e descompressão de dados *JPEG* e *MPEG*.

4.3 SISTEMA GRÁFICO

O sistema gráfico do *Playstation* é composto por três partes principais: o motor de transformações geométricas (GTE – *Geometry Transformation Engine*), pela unidade de processamento gráfico (GPU – *Graphic Processing Unit*), e por um *frame buffer* de 1MB.

O GTE é um veloz co processador matemático anexado ao processador principal. Este co processador provê vários serviços implementados em nível de hardware, tais como cálculos para rotação, translação e perspectiva, transformação de pontos, fontes de luz, efeitos de névoa (*fogging*), cálculos para a visualização de superfícies (*depth cueing*), interpolação linear e várias funções para vetores e matrizes. Segundo Loser (2001) muitos desses serviços são mais rápidos do que simples multiplicações e divisões no processador principal. É importante distinguir o GTE do GPU, o primeiro é responsável pelos cálculos matemáticos envolvidos na geração de uma imagem bidimensional a partir de um ambiente tridimensional, já o segundo é responsável pelo desenho de polígonos na imagem.

A GPU é um motor de renderização gráfica especializada, de alta velocidade, e pode utilizar os resultados dos cálculos da GTE em seus comandos. A GPU desenha gráficos na área de desenho do *frame buffer*, executando primitivas armazenadas na memória principal. A GPU suporta dois modos de exibição de cores, o modo direto 15 bits, que permite a exibição simultânea 32.768 cores, e o modo direto 24 bits que permite simultaneamente 16.777.216 de cores. Entretanto, no modo direto 24 bits, apenas imagens que tenham sido enviadas ao *frame*

buffer podem ser visualizadas (exibição de imagens imóveis), e a execução das funções de desenho da GPU não é possível. A GPU renderiza 360.000 polígonos por segundo, incluindo mapeamento de textura, efeitos de iluminação e tonalidades (Sony Computer Entertainment Inc. 1997b). O quadro 9 mostra os modos de vídeos suportados pela GPU (Sony Computer Entertainment Europe 1999a).

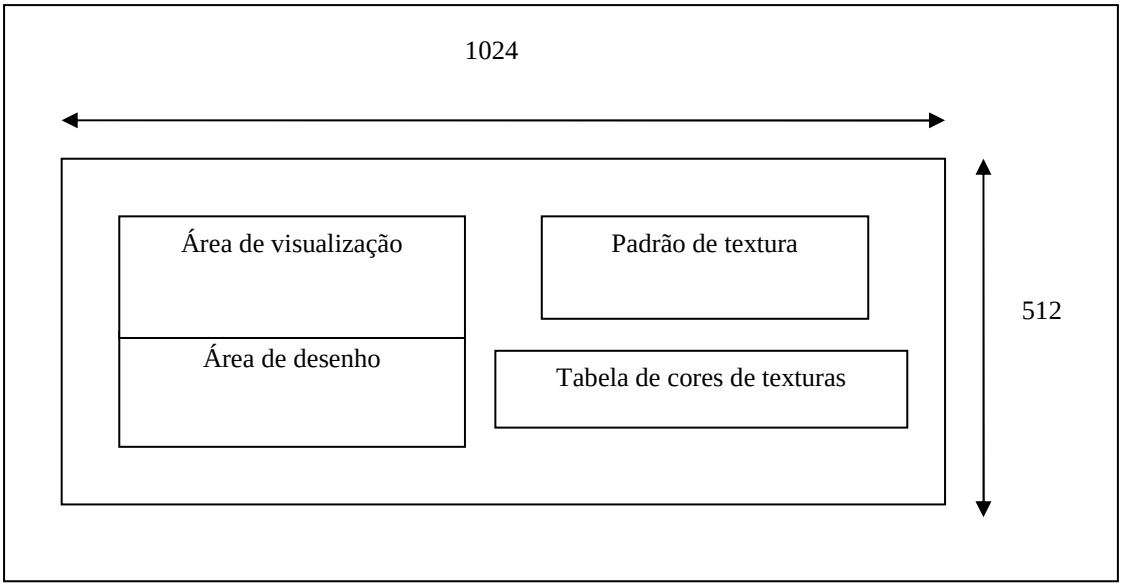
Quadro 9 – Modos de vídeo.

NTSC		*PAL*	
Entrelaçado	Não entrelaçado	Entrelaçado	Não entrelaçado
256(H) X 480(V)	256(H) X 240(V)	256(H) X 512(V)	256(H) X 256(V)
320 X 480	320 X 240	320 X 512	320 X 256
512 X 480	512 X 240	512 X 512	512 X 256
640 X 480	640 X 240	640 X 512	640 X 256
384 X 480	384 X 240	384 X 512	384 X 256

Fonte: Sony Computer Entertainment Inc. (1997b)

A área de memória de vídeo denominada *frame buffer*, é utilizada para armazenar dados gráficos, incluindo informações utilizadas pela imagem de vídeo atual, uma área de desenho, assim como tabelas de cores e texturas. O *frame buffer* (figura 23) é organizado como um *bitmap* de 1024 pixels de largura por 512 de altura com 16 bits de resolução de cor por pixel. Essa memória não é tratada linearmente, mas é acessada por coordenadas, tendo como sua origem o seu canto superior esquerdo. O *frame buffer* não pode ser acessado diretamente pelo processador principal, apenas pela GPU (Sony Computer Entertainment Europe 1999a).

Figura 23 – *Frame buffer*.



Fonte: Adaptado de Sony Computer Entertainment Inc.(1997b)

Dados lidos do *frame buffer* são continuamente utilizados para gerar o sinal de vídeo exibido no monitor. Para evitar que a área de desenho coincida com a área de visualização, utiliza-se a técnica *double buffering*. Nesta técnica, são preparadas duas áreas de mesma dimensão no *frame buffer*. Enquanto uma área recebe as instruções de desenho, a outra é exibida. Quando a área de desenho estiver pronta, as duas áreas são trocadas.

O sistema de som do Playstation é composto pela Unidade de Processamento Sonoro (SPU - *Sound Processing Unit*) e pelo decodificador de CD-ROM.

A SPU possui 24 canais de voz, controla 512 KB de memória conhecida como *sound buffer*, que não pode ser acessada diretamente pela CPU, possui suporte a MIDI e efeitos digitais como e*nvelope*, *looping* e *reverb*. Dados compactados em áudio digital são armazenados no *sound buffer*, sendo então processados pela SPU a uma taxa de amostragem de 44,1 KHz (Sony Computer Entertainment Inc. 1997b).

Dados compactados providos por CD-ROM XA ou dados CD-DA 16 bit, *PCM* são reproduzidos diretamente à saída de áudio, ou transmitidos à memória principal. Já os dados lidos do CD-ROM e gravados no buffer do CD-ROM são processados pelo decodificador do CD-ROM, e o sinal de áudio resultante é enviado à SPU.

A unidade de CD-ROM possui taxa de transferência de 150 KB/s na velocidade simples e 300 KB/s na velocidade dupla, suporte a CD de áudio e buffer de memória de 32KB.

O dispositivo controlador transmite as intenções do jogador à aplicação. Além dos dois conectores disponíveis no console, é possível conectar um acessório de expansão chamado de *multi tap*, onde podem ser conectados mais dispositivos controladores.

O cartão de memória permite que informações sobre o jogo sejam gravadas para serem utilizadas posteriormente. O cartão original possui 128KB que são divididos em blocos de 8KB, totalizando 16 blocos de dados, sendo que 15 blocos estão disponíveis para o jogador e 1 bloco é utilizado como sistema de arquivos. Como acontece com os controladores, além dos dois conectores padrão, outros podem ser conectados utilizando o *multi tap* (figura 24).

Figura 24 – Cartão de memória e *multi tap*.

O *Playstation* possui duas portas de expansão, uma serial e outra paralela. A porta serial é utilizada para conectar dois consoles, cada um ligado a um aparelho de televisão, podendo ser utilizados para que dois jogadores joguem em equipe ou como adversários. Originalmente a porta paralela ficou reservada para futuras expansões, e nos modelos mais recentes, essa porta não está mais disponível (Sony Computer Entertainment Inc. 1997b).

4.4 AMBIENTES DE DESENVOLVIMENTO

A única alternativa oficial para o desenvolvimento amador para o *Playstation* é o sistema *Net Yaroze* (Obiwahn 2000). Com a utilização de alguns acessórios não oficiais e customizados, ou até mesmo montados pelo próprio programador, é possível executar a partir de um microcomputador, programas compilados para o *Playstation*. A seguir serão vistas as principais alternativas de hardware e software para o desenvolvimento amador e acadêmico para o *Playstation*, sendo que para desenvolver profissionalmente ou comercialmente é necessária obtenção de uma licença e equipamentos oficiais (Loser 2001).

4.4.1 NET YAROZE

Lançado pela *Sony Computer Entertainment Inc*. em 1996, o projeto *Net Yaroze* possibilitou que programadores amadores criassem aplicativos para o *Playstation* utilizando ferramentas de desenvolvimento em um microcomputador conectado a um console especial, o *Playstation Net Yaroze* (figura 25) (Sony Computer Entertainment Inc. 1997a).

Figura 25 – *Playstation Net Yaroze*.

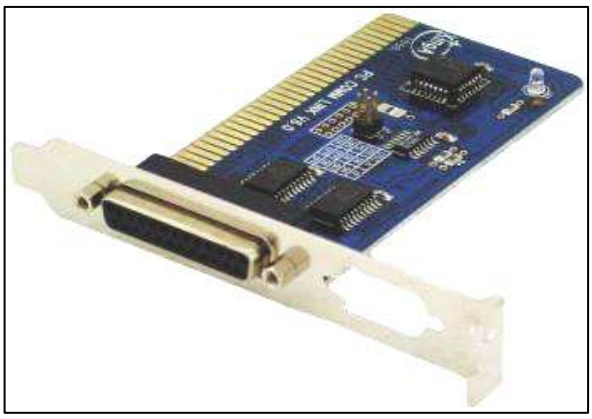
O console de cor negra era conectado a um microcomputador PC por um cabo serial. Fazia parte ainda do sistema um cartão de acesso, CDs contendo o compilador (C), bibliotecas

e diversos utilitários, além de acesso a suporte *online*. Segundo Homebrew (2001), o projeto tinha alguns obstáculos, o principal deles era o alto custo de aquisição. Foram vendidos em número limitado e os membros poderiam distribuir seu projetos apenas para outros membros. O acesso a alguns acessórios e funcionalidades do *Playstation*, pelo sistema *Net Yaroze* era limitado. Além disso, era necessário carregar uma biblioteca de funções na memória do console, o que limitava o seu uso.

4.4.2 AMBIENTES DE DESENVOLVIMENTO ALTERNATIVOS

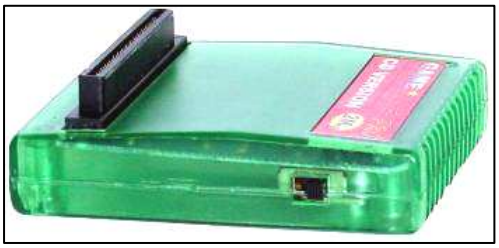
Uma empresa inglesa, a *Datel Design & Development Ltd*. (Homebrew 2001), desenvolveu nesta época um cartucho para ser conectado à porta paralela do console, o *Action Replay* (AR) ou *Game Shark* (GS) na América do Norte. Esse acessório permitia editar muitas características dos jogos, tais como avançar estágios, receber invencibilidade e munição ilimitada, ou seja, facilitar o jogo. Para isso, códigos previamente gravados eram selecionados, e através de uma placa de interface ISA 8 bits chamada de *Comms Link* (figura 26) conectada a um microcomputador, poderia ser conectada ao AR para carregar novos códigos.

Figura 26 – Placa *Comms Link*.



Atualmente a *Datel Design & Development Ltd*. não fabrica mais o AR, mas segundo Psxdev, podem ser utilizados para o mesmos propósitos, outros dispositivos similares fabricados na China (figura 27).

Figura 27 – Dispositivo similar ao *Action Replay*.



Alguns programadores conhecidos na comunidade de desenvolvimento de jogos por pseudônimos como Blackbag, Hitmen, Snake & McBain, desenvolveram um software chamado *Ez-o-ray*, que após gravado na memória do AR, permitia executar aplicativos no *Playstation* a partir do *PC*, utilizando a conexão com o AR e *Comms Link* (Homebrew 2001). Mais tarde Yumms desenvolveu o *Caetla*, que se tornou o mais popular programa para a reposição da memória do AR, para o desenvolvimento amador de jogos para o *Playstation* (Homebrew 2001).

Os compiladores C mais utilizados para o desenvolvimento amador e acadêmico para o *Playstation* são o *Psy-Q* e o *Yaroze*, que segundo (Loser 2001) é uma versão restrita do *Psy-Q*. Outras opções de compiladores C incluem o *GNU GCC* e o *Code Warrior*, além de outros compiladores para linguagens *Assembly* e *BASIC*. Existe ainda um outro cartucho semelhante ao AR, o *Xplorer* ou *Xploder* na América do Norte (figura 28). A principal diferença é que este cartucho é conectado diretamente na porta paralela do *PC*, dispensando assim, o uso da placa *Comms Link*. O *Caetla* também pode ser gravado no *Xplorer*, e suas recentes versões tem como principais características, a possibilidade de gerenciamento dos cartões de memória a partir do *PC*, busca e reprodução de arquivos de vídeo e som do CD ROM (Loser 2001).

Figura 28 – *Xploder*.

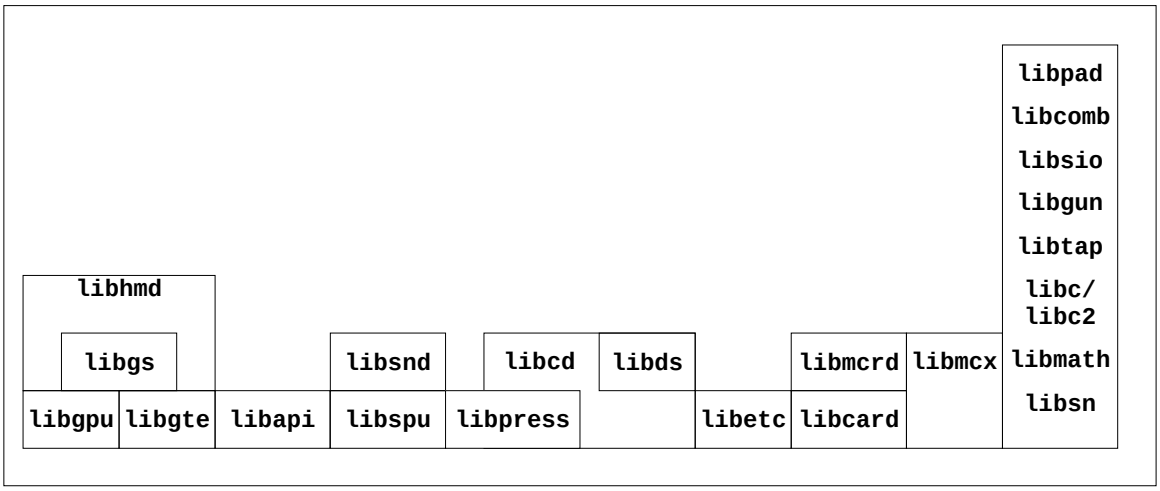
Outra solução para a conexão entre o *PC* e *Playstation* é a montagem de um cabo serial semelhante ao *Yaroze* conhecido como *Skywalker* (Bee [2001?]). Esta solução utiliza o software de comunicação *Siocons* juntamente com o disco de *boot Net Yaroze* (Hitmen 2000). Segundo Homebrew (2001), um emulador do *Playstation* no *PC* pode ser utilizado para testes e depuração. Atualmente, o emulador mais utilizado pela comunidade de desenvolvimento acadêmico para *Playstation* é o *PSEmu Pro*. Além do *hardware* e compiladores acima citados, são necessários utilitários, para a criação, edição e conversão de arquivos de som, vídeo, imagem e modelos tridimensionais.

4.5 COMPILADOR PSY-Q

Desenvolvido pela *SN Systems* o *Psy-Q* é o compilador oficial da *Sony Corporation* para desenvolvimento de jogos para o *Playstation*, utilizado profissionalmente. Este compilador é também o mais utilizado para o desenvolvimento amador.

Fazem parte do ambiente *Psy-Q*, bibliotecas que podem ser tanto de baixo nível quanto de alto nível, dependendo de seu relacionamento com o *PSX OS* (Sony Computer Entertainment Europe 1999a). Elas formam uma estrutura de dois níveis. Programas podem utilizar os níveis como necessário, inclusive utilizando os dois níveis de forma concorrente, com algumas exceções. A figura 29 mostra a estrutura das bibliotecas para o *Playstation*.

Figura 29 – Componentes da biblioteca do *PSX OS*.



Fonte: Sony Computer Entertainment Europe (1999a).

A seguir são descritas as principais caracteristicas de cada biblioteca:

a) *libapi* (biblioteca do núcleo), provê uma interface entre o sistema operacional e aplicações;
b) *lib*/*libc2* (biblioteca padrão C), subconjunto da biblioteca padrão C, incluindo funções de manipulação de caracteres, operação de memória e *jumps* não locais;
c) *libmath* (biblioteca matemática), contém funções matemáticas *ANSII IEEE754* e pacote de software para cálculo de número reais;
d) *libcard* (biblioteca do cartão de memória), provê funções para o controle do cartão de memória;
e) *libmcrd* (biblioteca do cartão de memória estendida), provê interface de alto nível ao cartão de memória;
f) *libpress* (biblioteca de compressão de dados), provê funções para a compressão e descompressão de dados de som e imagem;
g) *libgpu* (biblioteca gráfica básica), contém os comandos para construção de listas de desenho e manipulação de dados para entidades como polígonos, linhas e *sprites*;
h) *libgte* (biblioteca geométrica básica), biblioteca para a manipulação de dados como matrizes e vetores;
i) *libgs* (biblioteca gráfica estendida), biblioteca para gráficos tridimensionais, utiliza a *libgpu* e a *libgte*. Manipula entidades mais complexas como objetos e superfícies de fundo;
j) *libcd* (biblioteca do CD), lê dados de programas, imagens e som do *drive* de CD-ROM e executa áudio digital;
k) *libds* (biblioteca do CD estendida), possui as mesmas capacidades da *libcd* além de realizar recuperação de erro;
l) *libetc* (biblioteca de periféricos), realiza controle de *callbacks* para o uso de controladores e outros dispositivos periféricos além do processamento em baixo nível de interrupções;
m) *libtap* (biblioteca do *multi tap*), possibilita o acesso a vários controladores e cartões de memória conectados ao *multi tap*, que permite que sejam conectados de três a oito desses dispositivos;

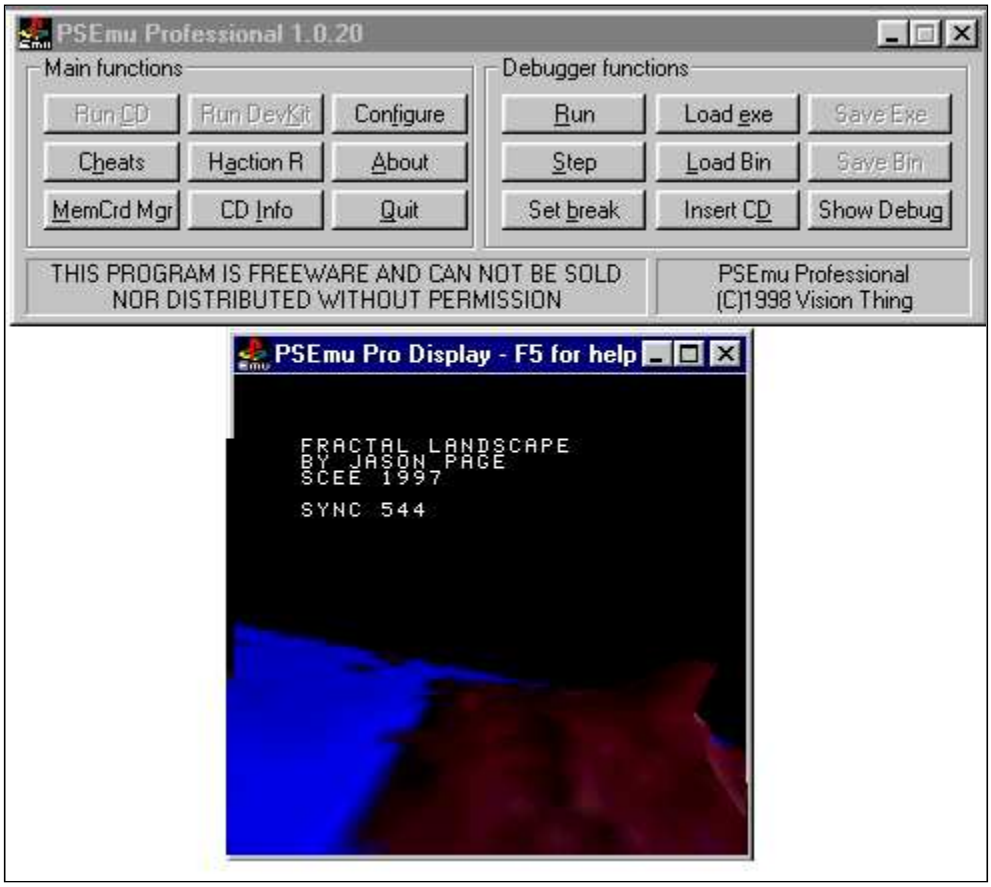
n) *libgun* (biblioteca da pistola), provê acesso ao uso de acessórios do tipo apontadores luminosos, como pistolas, que podem ser conectadas no conector do dispositivo controlador;
o) *libpad* (biblioteca do controlador), biblioteca para o acesso à controladores, possui protocolos para controladores extendidos como o *Dual Shock*;
p) *libcomb* (biblioteca do cabo *link*), biblioteca para o uso do cabo *link* de interface serial, que permite a conexão de dois consoles;
q) *libsnd* (biblioteca de som estendida), provê acesso a funções que permitem a execução de arquivos de som, lidos previamente;
r) *libspu* (biblioteca de som básica), controla a SPU;
s) *libsio* (biblioteca da entrada e saída serial), permite o uso da porta serial;
t) *libhmd* (biblioteca HMD), provê funções e definições para a manipulação do formato HMD, que integra modelagem, animação, textura e dados *MIMe*;
u) *lib* (biblioteca PDA), provê o acesso ao PDA quando conectado à entrada do cartão de memória;
v) *mcgui* (módulo de interface gráfica com o usuário para o cartão de memória), módulo que provê suporte para a recuperação e o salvamento de dados de jogos, assim como o suporte para a interface ao usuário.

Além dessas bibliotecas, fazem parte do ambiente *Psy-Q* vários utilitários, entre eles conversores e editores de imagens *raster*, modelos tridimensionais e arquivos de áudio, assim como um depurador. Outros utilitários desenvolvidos por programadores amadores podem ser encontrados em diversos *sites* sobre desenvolvimento de jogos para o *Playstation*.

4.6 EMULADOR

Atualmente diversos emuladores de *Playstation* para o PC estão disponíveis. Entre os mais populares encontram-se o *AdriPSX*, *Bleem!*, *ePSXe*, *PSEmu Pro* e *Virtual Game Station*. Segundo Homebrew (2001), o *PSEmu Pro* (figura 30) é o melhor emulador para o testes e depuração.

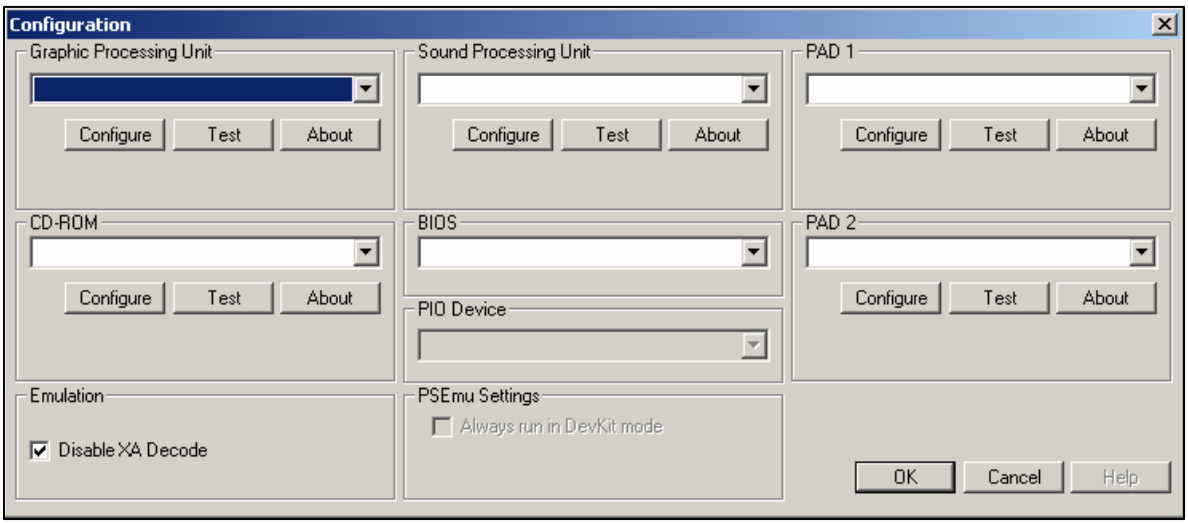
Figura 30 – Emulador *PSEmu Pro* em modo janela.



Apesar de não receber atualizações desde fevereiro de 1999, novos *plugins* para o *PSEmu Pro* têm sido lançados por programadores não ligados ao desenvolvimento deste emulador, o que permite configurá-lo para diferentes máquinas. *O PSEmu Pro* utiliza diversos *plugins* para simular os diferentes componentes de hardware do *Playstation*, tais como o leitor de CD-ROM, a SPU, a GPU e os dispositivos controladores. Para cada um desses componentes é possível encontrar diversas versões de *plugins.* Dessa forma é possível otimizar o uso dos recursos específicos de cada dispositivo do computador como placas de

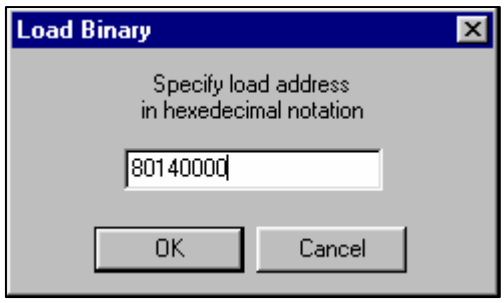
vídeo (*Open GL*, *DirectX*, *3dfx*, *Glide*[3]), leitores de CD-ROM, placas de som e dispositivos de entrada (controladores, *joystics*, teclados e o *mouse*). Além destes é necessária uma imagem da *bios* do *Playstation* (Homebrew 2001). A figura 31 mostra a tela de configuração dos *plugins* do *PSEmu Pro*.

Figura 31 – Configuração dos *plugins*.



Para executar programas no *PSEmu Pro*, é necessário carregar além do arquivo executável, todos os demais arquivos acessados pelo aplicativo, como som, gráficos e modelos, especificando-se o endereço de memória a serem carregados (figura 32).

Figura 32 – Diálogo de especificação do endereço de memória.



Para a depuração, o *PSEmu Pro*, permite visualizar o conteúdo dos registradores do processador e coprocessadores (figura 33), assim como o conteúdo da memória (figura 34).

Figura 33 – Registradores.

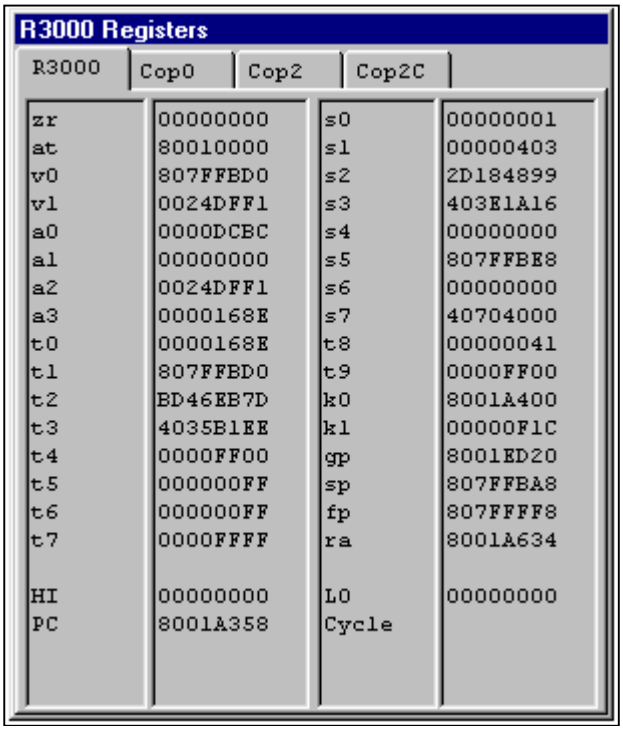
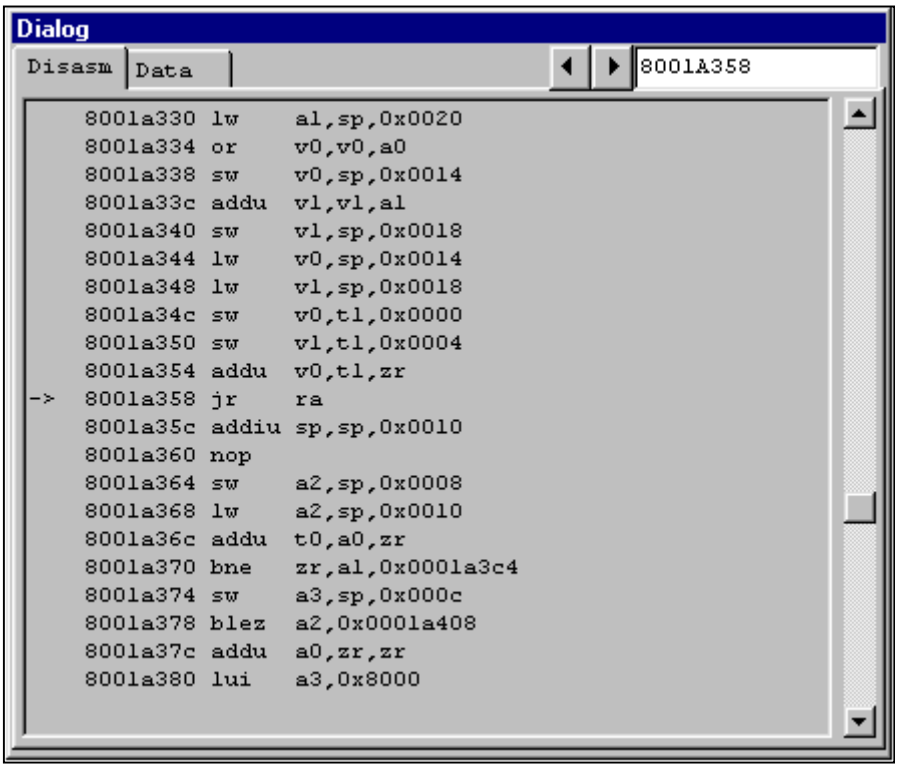


Figura 34 – Memória.



5 PROTÓTIPO

Apresenta-se neste capítulo questões relativas à especificação do protótipo, sua implementação e suas principais características.

5.1 REQUISITOS DO PROTÓTIPO

O protótipo desenvolvido demonstra o funcionamento e uso de árvores *BSP* para determinação de superfícies visíveis na plataforma *Playstation.* Para tal, é necessário que o protótipo ofereça algum nível de interatividade com o usuário, procurando simular um jogo interativo. O presente protótipo procura realizar esta tarefa implementando um personagem, representado graficamente por um triângulo, comandado interativamente pelo usuário, que se desloca em um cenário bidimensional.

Para a especificação do protótipo, utilizou-se a análise estruturada pois a linguagem na qual foi desenvolvido o protótipo, a linguagem C, não é orientada a objetos. Para a especificação do protótipo utilizou-se o *PowerDesigner Process Analyst* (diagrama de contexto) e o *Flow4* (fluxograma). Utilizou-se o compilador e ferramentas do ambiente *PSy-Q*, da *SN Systems Ltd.* e ainda o editor de textos *Wordpad*. Para realizar testes do funcionamento de algumas funções durante o desenvolvimento do protótipo, foi utilizado o ambiente de desenvolvimento *Visual C++ 6.0*, da *Microsoft*.

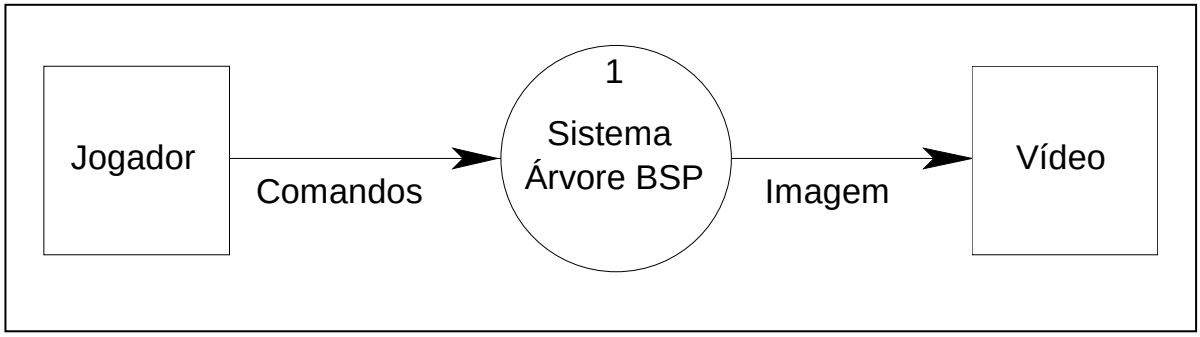
Devido às dificuldades encontradas para a conexão do *Playstation* ao PC, os testes do protótipo foram realizados com o emulador *PSEmu Pro*, em um microcomputador *Pentium* 200 MHz, com 64MB de memória principal e placa de vídeo com aceleração 3D, com 4MB de memória, sendo que algumas versões do protótipo foram gravadas em CD para a execução e testes no *Playstation*.

5.2 ESPECIFICAÇÃO

No diagrama de contexto do protótipo (Pressman 1995), apresentado pela figura 35, os comandos necessários para a interação entre o jogador e o protótipo são fornecidas através do controlador, o dispositivo de entrada do *Playstation*, semelhante a um *joystick*. O sistema realiza os cálculos e testes necessários para a ordenação das faces, a partir dos dados

informados pelo jogador. O sistema gera as informações necessárias para a renderização dos dados na tela de vídeo.

Figura 35 – Diagrama de contexto.

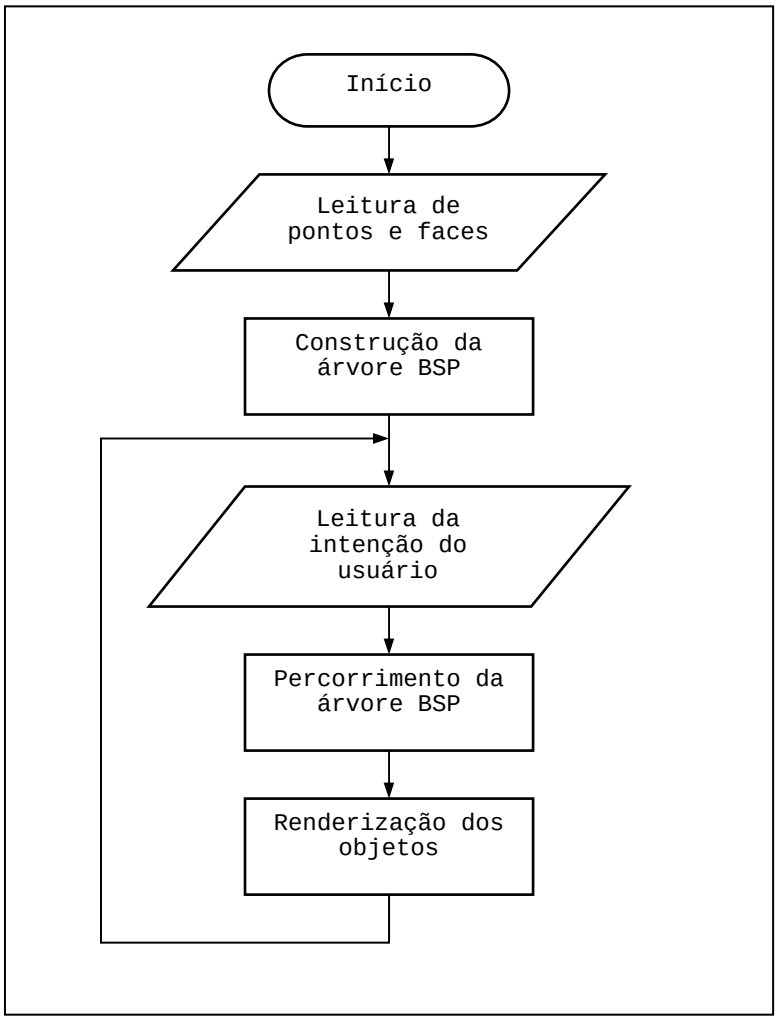


5.2.1 FLUXOGRAMA DO PROTÓTIPO

A figura 36 mostra a ordem de execução das funções do protótipo. Após o início do programa, são lidos os pontos e faces do cenário.

Durante a construção da árvore *BSP*, são realizados alguns testes. Dentre eles a verificação do posicionamento de uma face em relação a um hiperplano. Se o resultado deste teste apontar uma interseção, a face é dividida. Após a árvore ser construída, são lidos os dados do controlador, que representam as intenções do usuário, que poderá mover o ponto de vista do observador, alterar o passo do observador ou determinar o fim da execução do protótipo.

Figura 36 – Fluxograma do protótipo.



5.3 IMPLEMENTAÇÃO

Apresenta-se neste item a definição de estruturas utilizadas no protótipo assim como alguns algoritmos e a operacionalidade do protótipo desenvolvido.

5.3.1 ESTRUTURAS DE DADOS

Durante a leitura de dados, pontos e faces, três estruturas são utilizadas. A *Point2* armazena as coordenadas x e y, para a definição de um ponto em duas dimensões, ambas do tipo real. Duas estruturas do tipo lista são definidas. *ListaPontos* armazena pontos e *ListaFaces* armazena as informações necessárias sobre as faces ou arestas que formam o cenário em duas dimensões, os pontos inicial, final e um identificador. As listas são circulares

e duplamente encadeadas. O quadro 10 mostra o código fonte para a definição dos três tipos, além da estrutura genérica para a construção da lista circular, *Dll*.

Quadro 10 – Definição dos tipos *Point2*, *ListaPontos* e *ListaFaces*.

```
typedef struct Point2Struct
{
	double x, y;
} Point2;

typedef struct _listaPontos
{
	struct _listaPontos* next;
	struct _listaPontos* prev;
	int Id;
	Point2 P;
} ListaPontos;

typedef struct _listaFaces
{
	struct _listaFaces* next;
	struct _listaFaces* prev;
	int Id;
	int Pini;
	int Pfim;

} ListaFaces;

typedef struct _dllrec
{
	struct _dllrec *next;
	struct _dllrec *prev;
} *Dll, DllRec;
```

Para a construção da árvore *BSP* foi definida uma estrutura chamada *ArvoreBSP* (quadro 11), que é utilizada para armazenar as informações relativas à árvore binária (árvore *BSP*). Além dos dois ponteiro para os nodos filhos, frente (apontador esquerdo) e trás (apontador direito), há um ponteiro para uma lista de faces, que receberá as informações das faces coincidentes (colineares) em relação à sua posição no espaço.

Quadro 11 – Estrutura *ArvoreBSP*.

```
typedef struct _arvoreBSP
{
	struct _arvoreBSP* frente;
	struct _arvoreBSP* tras;
	ListaFaces* listaFaces;
} ArvoreBSP;
```

5.3.2 ALGORITMOS

Abaixo são demonstrados alguns algoritmos, em código fonte, utilizados pelo protótipo. O quadro 12 mostra o código fonte para a inclusão de um novo elemento na lista circular genérica.

Quadro 12 – Inclusão de um elemento na lista.

```
Dll DllAddEnd( Dll *head, unsigned int size )
{
	Dll nnew;
	nnew = (Dll)calloc( 1, size );// aloca memória para o novo elemento
	if( nnew == NULL ) // se não conseguiu retorna nulo
		return( NULL );
	if( *head == NULL )
	{
		// se chegou aqui, a cabeça da lista é nula, e este é o primeiro elemento
		nnew->next = nnew;
		nnew->prev = nnew;
		*head = nnew;
	}
	else
	{
		// não é o primeiro a ser incluido na lista
		nnew->next = *head;
		nnew->prev = (*head)->prev;
		(*head)->prev->next = nnew;
		(*head)->prev = nnew;
	}
	return( nnew );
}
```

O percorrimento (de forma recursiva) da árvore *BSP* e a renderização de seus objetos é feita pela função *PercorreArvore* (quadro 13). Nota-se que a renderização não é realmente feita, mas sim a inicialização de uma matriz do tipo primitiva *LINE_G2*, que recebe os valores dos pontos inicial e final, e a informação relativa à cor (padrão *RGB*) para cada ponto. O tipo primitiva *LINE_G2*, aceita a atribução de cores diferentes para os seus pontos inicial e final (Sony Computer Entertainment Europe 1999b). A estrutura *LINE_G2* é nativa da biblioteca *libgpu*.

Quadro 13 – Percorrimento da árvore *BSP*.

```
void PercorreArvore(ArvoreBSP* pArvoreBSP, double x, double y)
 {
        //inicialização das variáveis
        char *cBuffer;
        short i = 0;
        ListaFaces* pAux = NULL;
        ListaFaces* pTemp = NULL;
        Point2 p2FaceIni, p2FaceFim;
        int iResultado = -1;
        if (pArvoreBSP == NULL)
              return;
        p2FaceIni = BuscaPonto(pArvoreBSP->listaFaces->Pini); //busca dados do ponto
        p2FaceFim = BuscaPonto(pArvoreBSP->listaFaces->Pfim); //busca dados do ponto
        iResultado = VerificaPosicao(p2FaceIni.x, p2FaceIni.y, p2FaceFim.x, p2FaceFim.y, x,
       y, x, y); //teste da posição do observador em relação à face atual
        switch (iResultado)
        {
        case FRENTE:
             PercorreArvore(pArvoreBSP->tras, x, y); //percorre no sentido contrário
             pAux = &(*pArvoreBSP->listaFaces);
             pTemp = pAux;
             do // enquanto houver faces na lista de faces colineares permanece neste laço
             {
                 if (iConta > 0)
                 {
                     p2FaceIni = BuscaPonto(pAux->Pini); //busca dados do ponto
                     p2FaceFim = BuscaPonto(pAux->Pfim); //busca dados do ponto
                 }
                 FntPrint("%d\n", pAux->Id); //função para escrever na stream de impressão
                 g2[iConta].x0 = (short) p2FaceIni.x; //atribuição dos valores dos pontos da
                 g2[iConta].y0 = (short) p2FaceIni.y; //face à estrutura g2
                 g2[iConta].x1 = (short) p2FaceFim.x;
                 g2[iConta].y1 = (short) p2FaceFim.y;
                 setRGB0(&g2[iConta], 0, 0, 255); //atribui a cor azul para o ponto inicial
                 setRGB1(&g2[iConta], 255, 255, 255); //atribui a cor branca para o ponto final
                 pAux = pAux->next;
                 iConta++; //incremento do contador de faces
             } while (pTemp != pAux);
             PercorreArvore(pArvoreBSP->frente, x, y);
             break;
        default:
             PercorreArvore(pArvoreBSP->frente, x, y); //percorre na ordem contrária
             pAux = &(*pArvoreBSP->listaFaces);
             pTemp = pAux;
             do // enquanto houver faces na lista de faces colineares permanece neste laço
             {
                 if (iConta > 0)
                 {
                     p2FaceIni = BuscaPonto(pAux->Pini);
                     p2FaceFim = BuscaPonto(pAux->Pfim);
                 }
                 FntPrint("%d\n", pAux->Id); //função para escrever na stream de impressão
                 g2[iConta].x0 = (short) p2FaceIni.x; //atribuição dos valores dos pontos da
                 g2[iConta].y0 = (short) p2FaceIni.y; //face à estrutura g2
                 g2[iConta].x1 = (short) p2FaceFim.x;
                 g2[iConta].y1 = (short) p2FaceFim.y;
                 setRGB0(&g2[iConta], 255, 0, 0); //atribui a cor vermelha para o ponto inicial
                 setRGB1(&g2[iConta], 255, 255, 255); //atribui a cor branca para o ponto final
                 pAux = pAux->next;
                 iConta++; //incremento do contador de faces
             } while (pTemp != pAux);
             PercorreArvore(pArvoreBSP->tras, x, y);
             break;
        }
 }
```

A função *DisplayAll* (quadro 14) atualiza os dados na tela, desenhando o conteúdo do *frame buffer*, incluindo a renderização das faces, que é feita a partir da leitura da matriz de faces.

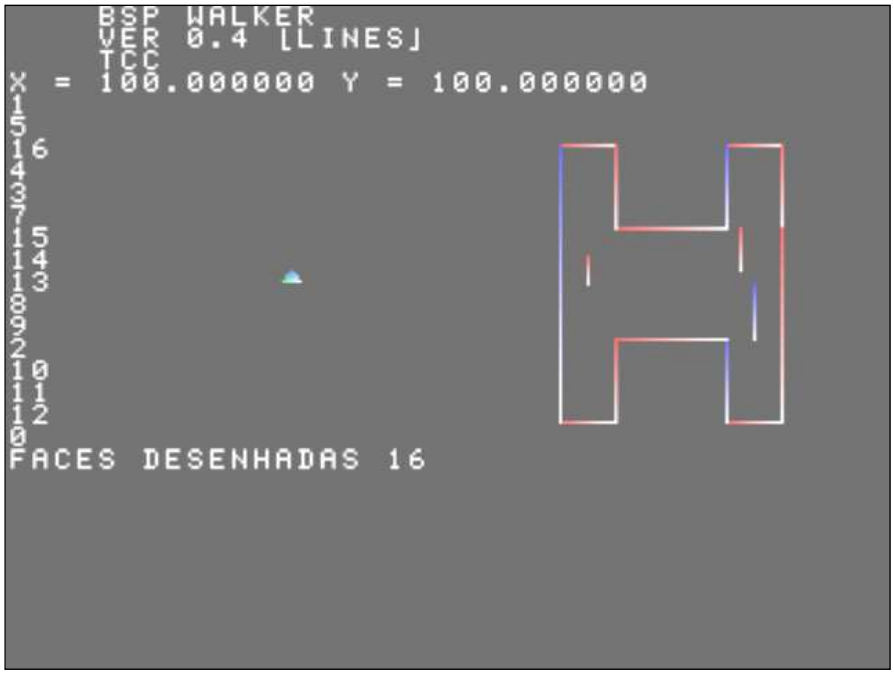
Quadro 14 – Função para a atualização da tela

```
void DisplayAll(int activeBuffer)
{
        short i = 0;
        DrawPrim(&g3); //função para executar a primitiva cujo endereço é passado como parâmetro
        for (i = 0; i < iConta; i++)
                DrawPrim(&g2[i]); //laço para executar a as primitivas armazenadas no vetor g2
        if (iConta > 0)
                FntPrint("faces desenhadas %d\n", --iConta); //função para escrever na stream de
impressão
        DrawSync(0); //função que aguarda o término das rotinas de desenho ou execução de primitivas
        VSync(0); //agurda o retorno da interrupção vertical
        GsSwapDispBuff();//troca a área de desenho pela área de visualização
        GsSortClear(0,0,0,&myOT[activeBuffer]);//executa um comando de limpeza de tela na tabela de
ordenação
        GsDrawOt(&myOT[activeBuffer]); //processa os comandos da GPU armazenados na tabela de
ordenação
        FntFlush(-1);//desenha o conteúdo da stream de impressão no frame buffer
        iConta = 0;
}
```

5.3.3 OPERACIONALIDADE DA IMPLEMENTAÇÃO

Após a inicialização do protótipo com a leitura dos pontos e das faces e a construção da árvore *BSP*, é definido um ponto de vista arbitrário, representado por um triângulo. O cenário é então desenhado. Na extremidade esquerda da tela, é exibido o nome das faces na ordem em que foram desenhadas. Na extremidade direita é desenhado o cenário. Note-se que as faces são desenhadas com as cores inicial e final distintas, possibilitando assim a visualização das faces que foram divididas por terem interseção com um hiperplano. O valor de *faces desenhadas* representa o total de faces no cenário. A figura 37, mostra a tela inicial do protótipo.

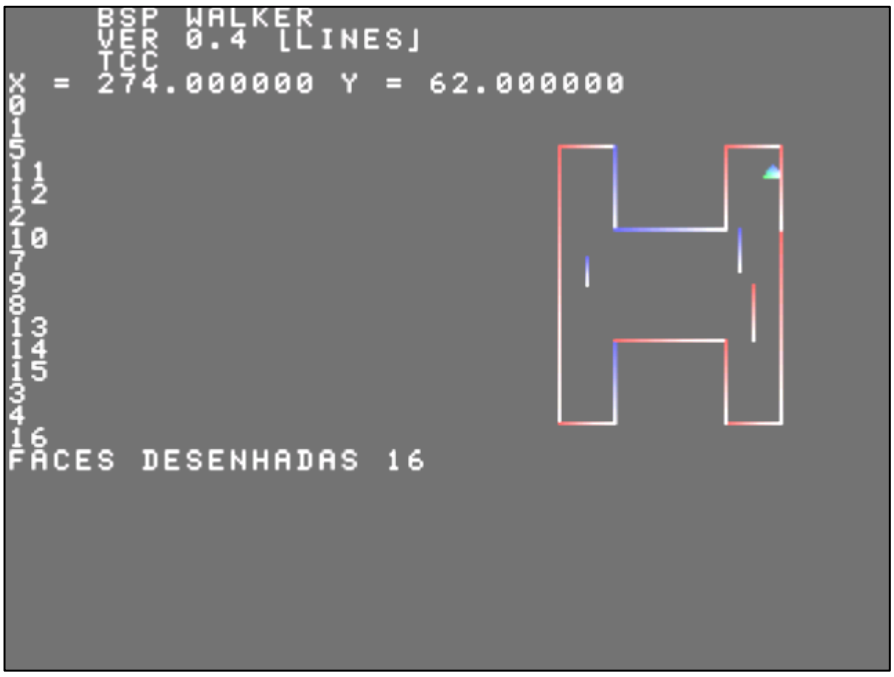
Figura 37 – Tela inicial do protótipo



Constantemente, o protótipo lê do dispositivo controlador as informações que definem as ações do jogador. São possíveis as movimentações em todas as direções no plano, através dos botões direcionais do controlador. O botão X, incrementa em uma unidade o passo do ponto de vista do observador, e o botão representado por um triângulo o decrementa. O valor inicial do passo é 4.

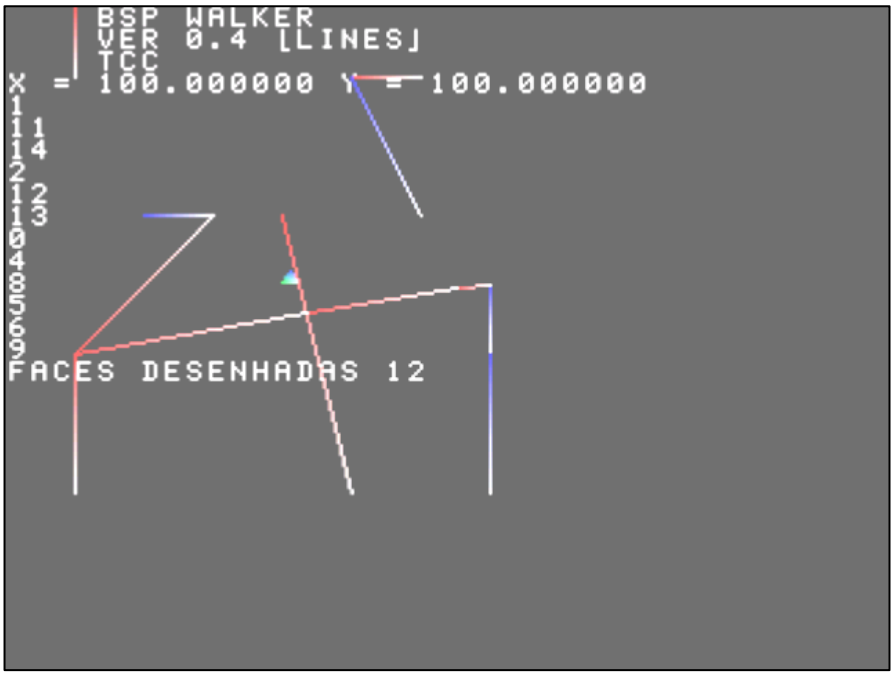
Após mover o ponto de vista do observador, a árvore é percorrida e é gerada uma nova ordenação para a renderização das faces do cenário. As cores das faces definem se o observador está situado à frente da face, cor vermelha, ou se está situado atrás da face, cor azul. A figura 38, mostra uma tela do protótipo onde o ponto de vista do observador foi alterado. Os valores de X e Y, representam as coordenadas do ponto de vista.

Figura 38 – Ponto de vista do observador alterado.



Na figura 39, é apresentado um diferente cenário. Nota-se um número maior de divisões das faces, geradas pelas interseções entre elas.

Figura 39 – Protótipo com um outro cenário.



6 CONCLUSÕES

O presente trabalho apresentou o desenvolvimento de um protótipo de software demonstrando o uso de árvores *BSP* em uma aplicação gráfica interativa, simulando um jogo de computador na plataforma *Playstation*.

O desenvolvimento deste trabalho mostrou-se muito satisfatório, pois permitiu o estudo e a aplicação de uma técnica de computação gráfica em um protótipo de jogo para um console cuja arquitetura é proprietária, e o acesso aos equipamentos de desenvolvimento e informações sobre a plataforma são restritos. Mesmo assim, foi possível, em caráter experimental, implementar um protótipo de software.

As ferramentas utilizadas satisfizeram as necessidades para o desenvolvimento do protótipo, porém os testes pretendidos por meio da conexão entre o microcomputador e o *Playstation* não foram possíveis. Acredita-se que alguns acessórios, em especial a placa de comunicação *Comms Link* e o *Game Hunter*, apresentam defeitos de projeto ou de fabricação visto que não são produtos oficiais suportados pelo fabricante do *Playstation*. Os testes realizados com o emulador permitiram visualizar e interagir com o protótipo, sem que houvesse a necessidade de gravá-lo em CD. A gravação do protótipo em CD fez-se necessária para alguns testes, pois, para que se obtenha bom desempenho com o emulador, são necessários recursos como placa de vídeo com aceleração 3D e processador de última geração.

6.1 EXTENSÕES

Como extensões a este trabalho, pode-se implementar as técnicas *back-face culling* e *view frustum culling*, a fim de otimizar o percorrimento da árvore *BSP*. Pode-se também modificar a estrutura da árvore *BSP*, com a finalidade de utilizá-la para a determinação de superfícies visíveis em cenários tridimensionais, assim como realizar a detecção de colisões, tanto em ambientes bidimensionais quanto em ambientes tridimensionais. Para tanto, será necessária a criação de novas rotinas para a renderização e o estudo da interatividade com o jogador. Pode-se estudar a aplicação de texturas aos cenários e a utilização de *sprites* para a representação de personagens em jogos tridimensionais.

Sugere-se ainda um estudo comparativo entre outras técnicas para a determinação de superfícies visíveis como o algoritmo *z-Buffer*, a utilização do algoritmo *scanline* com a árvore *BSP* e portais, tanto na plataforma *Playstation* quanto na plataforma PC.

Sob o aspecto do uso do hardware *Playstation*, pode-se utilizá-lo experimentalmente para outros tipos de aplicações totalmente diversas daquela para a qual ele foi projetado, a área de jogos eletrônicos. Um exemplo é seu uso como equipamento para controle e visualização de processos industriais, que pode ser abordado em futuros trabalhos de pesquisa.

REFERÊNCIAS BIBLIOGRÁFICAS

BATTAIOLA, André L. et al, Desenvolvimento de jogos em computadores e celulares. **RITA –** Revista de informática teórica e aplicada. SBC, Sociedade Brasileira de Computação, [S.l.], v. 8, n. 1, out. 2001.

BEE, Mike. **Skywalkers Playstation corner**. [S.l.]: [2001?]. Disponível em: < http://www.geocities.com/SiliconValley/Lab/6332/psx.html>. Acesso em: 03 dez. 2001.

CHIN. Norman. A walk trough bsp trees. **Computer graphics gems V**. Boston: AP Professional, 1995, 438p.

EBERLY, David H. **3D game engine design:** a practical approach to real-time computer graphics. San Francisco: Morgan Kaufmann, 2001. 560p.

FOLEY, James D. **Computer graphics:** principles and practice. 2. ed. Washington: Addison-Wesley, 1990. 1175p.

HITMEN. **Home of Hitmen**. [S.l.]: Apr. 2000. Disponível em: <http://www.hitmen-console.org>. Acesso em: 29 nov. 2001.

HOMEBREW Playstation development. [S.l.]: October 2001. Disponivel em: <http://www.psxdev.ip3.com>. Acesso em: 29 nov. 2001.

LAMOTHE, André. **Tricks of the Windows game programming gurus.** Indianapolis: Sams, 1999.

LOSER. **Loser´s psxdev page**. [S.l.]: February 2001. Disponível em: <http://www.loser-console.org/psx/>. Acesso em: 29 nov. 2001.

LEHMANN, Charles H.. **Geometria analítica.** 6. ed. Rio de Janeiro: Globo, 1987. 457p.

OBIWAHN. **Lucyforge.** [S.l]: June 2000. Disponível em: <http://www.myworld.privateweb.at/gavaman>. Acesso em: 29 nov. 2001.

SHIMER, Carl. **Binary space partition trees.** [S.l.]: 1997. Disponível em: <http://www.cs.wpi.edu/~matt/courses/cs563/talks/bsp/bsp.html>. Acesso em: 03 dez. 2001.

PRESSMAN, Roger S. **Engenharia de software**. Sao Paulo: Makron Books, 1995. 1056p.

SONY COMPUTER ENTERTAINMENT EUROPE. **Playstation operating system.** London, Aug. 1998. 30p.

SONY COMPUTER ENTERTAINMENT EUROPE. **Run-time library overview.** London, Aug. 1999a. 330p.

SONY COMPUTER ENTERTAINMENT EUROPE. **Run-time library reference.** London, Sept. 1999b. 1220p.

SONY COMPUTER ENTERTAINMENT EUROPE. **SCEE history.** London, 2001. Disponível em: <http://www.scee.com/corporate/sceehistory.jhtml>. Acesso em: 03 dez. 2001.

SONY COMPUTER ENTERTAINMENT INC. **Start up guide.** Tokyo, Feb. 1997a. 41p.

SONY COMPUTER ENTERTAINMENT INC. **User guide.** Tokyo, Feb. 1997b. 211p.

WADE, Bretton. **BSP tree frequently asked questions (FAQ).** Mountain View, Nov. 1997. Disponível em: <http://reality.sgi.com/bspfaq/whole.shtml>. Acesso em: 29 nov. 2001.