Universidade Federal de Sergipe
Centro de Ciências Exatas e Tecnologia
Departamento de Computação

Relatório

Profª. Beatriz Trinchão Andrade de Carvalho

Equipe:
Ícaro Marley Oliveira Fraga da Silva - 201210008538
Rodrigo Benedito Otoni - 201210009188
Thales Francisco Sousa Sampaio Alves dos Santos - 201210012648

São Cristóvão - SE
2016

Sumário

1. Introdução

O projeto foi feito durante a disciplina *Processamento de Imagens e Computação Gráfica* do *Departamento de Computação* da *Universidade Federal de Sergipe*, no semestre 2015-2.

Definido pela professora Beatriz, foi desenvolvido pelo grupo para acompanhar o aprendizado ao longo da primeira unidade da disciplina, cujo assunto foi *Computação Gráfica*.

A ideia principal foi montar um pequeno simulador e realizar sua impressão de dados em forma de imagens na tela, através da ferramenta *OpenGL*. A especificação completa do trabalho está detalhada no tópico seguinte deste relatório. Após isso apresentamos como se deu o desenvolvimento do projeto. A duas seções seguintes tratam do funcionamento de nossa implementação e de alguns exemplos de execução. Por fim apresentamos uma breve conlusão.

2. Especificação do Projeto

Esse trabalho é um simulador de ecossistema. Seu cenário consiste em um lago que contém peixes, pedras e plantas.Assim, vocês podem enxergar o lago como uma matriz 3D de posições, onde as pedras e plantas ocupam posições fixas, e os peixes nadam de posição em posição.

Esse lago tem suas três dimensões (comprimento, altura e largura) definidas por valores inteiros.

A primeira regra do lago da luta é... cada peixe nada em uma direção até encontrar alguma coisa. Essa coisa pode ser uma planta, outro peixe, uma pedra ou o fim do lago. Assim, caso o peixe:

• Encontre o fim do lago ou uma pedra: ele muda randomicamente para uma direção onde haja água;

• Encontre outro peixe, o maior come o menor. Em caso de empate, um dos peixes é escolhido aleatoriamente
como vencedor. O peixe vencedor cresce, acumulando a massa da presa.

• Passe por uma posição ocupada por uma planta, ele come parte da planta. O peixe cresce em massa, e a planta diminui.

A cada iteração do sistema, o peixe diminui (a não ser que coma algo) e as plantas crescem (a não ser que sejam comidas). As taxas de diminuição dos peixes e de crescimento das plantas serão fornecidas ao programa como entrada. Se a massa do peixe passar de um certo limite ele "explode", criando entre 13 e 26 peixes menores nas posições vizinhas à posição onde o peixe estava. A massa combinada dos peixes menores deve ser igual à massa do
peixe original. Se uma planta cresce demais ela também se divide, gerando de 12 a 27 plantas menores nas posições vizinhas à posição e na posição onde a planta estava. Assim como os peixes, a massa combinada das plantas menores deve ser igual à massa da planta original.

Caso a massa da planta ou do peixe chegue a zero, eles morrem e desaparecem do simulador.

Neste cenário, a câmera deverá se movimentar livremente e não é necessário tratar as colisões dela com os elementos do lago. Mais especificamente, a câmera poderá:

• Virar para cima, para baixo, para a esquerda e para a direita;

• Se movimentar para a frente e para trás na direção que estiver.

No programa, deve ser aberta uma janela para a cena criada. O usuário deve navegar por essa cena em primeira pessoa, usando teclado e/ou mouse, de forma que seja possível executar os movimentos citados acima. A posição inicial da câmera fica à escolha do grupo.

Os peixes não precisam ser animados (isso já seria um extra), mas eles devem estar de frente para a direção em que estão nadando e não podem ficar de cabeça pra baixo. Os peixes começam a nadar a partir de uma posição aleatória, e podem ocupar qualquer lugar na água que não contenha uma pedra. As plantas também devem ser inicializadas em
posições aleatórias e podem ocupar qualquer lugar que um peixe ocupa. As pedras afundam e podem ficar em cima de outras pedras (a cena começa já com as pedras no fundo, distribuídas em posições aleatórias).

O programa deve permitir ao usuário acelerar ou desacelerar a evolução do ecossistema.

Com isso, vocês podem observar as interações entre os elementos a curto e longo prazo e pensar em entradas que permitam que seu ecossistema sobreviva por mais tempo.

3. Desenvolvimento do Projeto

O progresso do projeto foi lento, mas constante. Ao longo dos dias o grupo encontrou as soluções para os problemas encontrados no desenvolvimento pesquisando na internet e recebendo conselhos dos orientadores (professora Beatriz e monitor Cláudio).

Para reduzir problemas recorrentes causados por falta de tempo entre os participantes do grupo, os encontros foram feitos por *Google Hangout*, o controle de versão foi feito tanto pelo *GitHub* quanto pelo *Gmail*. O relatório foi feito no *Google Docs*.

A linguagem utilizada foi *C/C++*, linguagem robusta de programação, que causou algumas dificuldades ao grupo, principalmente devido a falta de experiência do grupo com a linguagem.

Inicialmente, em conjunto com a linguagem, foi utilizada a IDE *Codeblocks* mas eventualmente abandonada por motivos de complexidade. O restante do desenvolvimento foi realizado utilizando-se editores de texto (*notepad++* e *Geany*) e console (*cmd shell* e *terminal*).

Para montar o simulador, foram utilizados conceitos de orientação a objetos, relacionando pedras, plantas, peixes e o cenário, que é uma matriz 3D.

Em relação aos Sistemas Operacionais utilizados, os alunos Thales e Ícaro utilizaram o *Windows 10* e Rodrigo utilizou o *Linux Mint 17.2*. Graças a portabilidade do GitHub e ao cuidado com as versões mantido pelo grupo, não houve empecilho em relação a diferenças entre os sistemas operacionais.

Alguns dos problemas recorrentes no desenvolvimento que foram solucionados ao longo do tempo:

- Erros de includes (undefined reference, multiple definitions)
- Erros de acesso indevido a memória (segmentation fault)
- Funcionamento da câmera (translação, rotação)

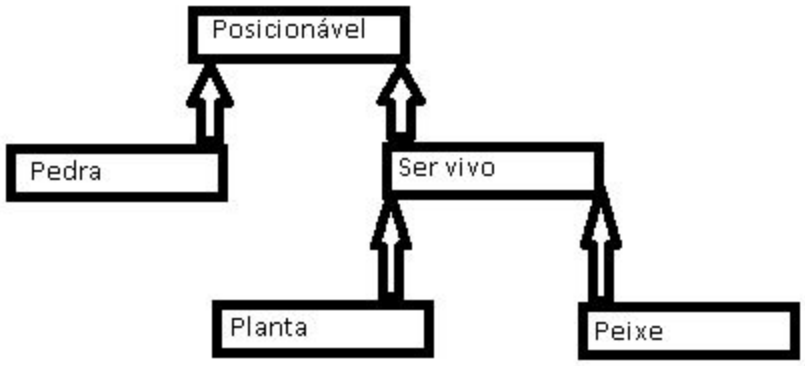
4. Funcionamento do Projeto

O projeto foi dividido em duas partes, os elementos do simulador e a parte de visualização, referente ao uso de *OpenGL*.

4.1. Simulador (Posicionavel.cpp, Ecossistema.cpp e main.cpp):

As classes foram criadas utilizando o conceito de herança, do paradigma orientado a objeto. A mais alta superclasse é chamada de *Posicionavel*, as classes *Pedra*, *Peixe* e *Planta* são filhas desta primeira. Ainda, a classe *Ser vivo* é superclasse de *Planta* e *Peixe*, sendo também filha de *Posicionável*.

A estrutura de classes desenvolvida pode ser vista a seguir:



As classes foram definidas dessa forma para melhor aproveitar a modularização de código. Abaixo, as declarações (no .h) de cada uma:

```
class Posicionavel
//pedra planta e peixe herdam dessa classe.
{
    protected:
        posicao localizacao;
        int id; //identificador de posicionavel
        void posicionar();
        //codigo para gerar uma posicao aleatoria x y z e alocar posicionavel no cubo
        //peixe e planta tem a mesma função, pedra sobrecarrega esta

    public:
        posicao* getPosicao();
        void setPosicao(int x,int y, int z);
        int getId();
        Posicionavel(int id);
        //construtor
        //seta id

};
```

```
class SerVivo: public Posicionavel
//pedra planta e peixe herdam dessa classe.
{
    protected:
        int massa; //massa do ser
        int limite; //limite de massa
        int taxa; //taxa de crescimento/decrescimento
        bool agiu; //marca se agiu nesse turno


    public:
        SerVivo(int massa,int taxa,int limite,int id);
        //passa id para classe posicionavel
        //seta massa, taxa iniciais e limites iniciais
        int getTaxa();
        int getMassa();
        void setMassa(int massa);
        bool getAgiu();
        void setAgiu();
        void explodir();
        //chama a funcao morrer e aloca varios peixes menores no Cubo
        void diminuir(int qtd);
        // diminui massa em qtd pontos por fome/mordida. se a massa chegar a 0, chama a funcaoo morrer
        bool aumentar(int qtd);
        // aumenta massa em qtd pontos por fome/mordida. se a massa chegar a maximo, chama a funcaoo explodir
        //retorna true caso exploda
        void morrer();
        //tira referencia no cubo e coloca na posicao do limbo, 0,0,0
        int sangrar();
        //limpa sua posicao no Cubo (seta para 0,0,0,id)
        void agir();
        //sobrecarregada por peixe e planta
};
```

```
class Pedra: public Posicionavel
{
    private:
        void posicionar();
        //pedras são sempre alocadas no chao
        //pedra acumula em cima das outras
    public:
        Pedra();
        //construtor. chama metodo posicionar.
};
```

```
class Planta: public SerVivo
{
    private:
        void crescer();
        //chama funcao aumentar de acordo com a taxa de crescimento
    public:
        Planta(int taxaInicial, int x, int y, int z, int massa);
        //construtor da explosão seta planta em x y z
        Planta(int taxaInicial);
        //construtor. chama metodo posicionar.
        //chama construtor de serVivo, passando a massa e taxa, e 1000 como limite
        //passa id para construtor de serVivo
        int sangrar();
        //ocorre ao ser mordido. chama a funco privada diminuir, recebe a massa retornada
        //retorna a quantidade que "sangrou"
        void agir();
        //acao da planta
        //chama metodo crescer
};
```

A matriz 3D é definida em outra classe, chamada *Ecossistema*, e consiste de *structs* definidos da seguinte forma:

```
//unidade minima do cubo.
typedef struct {
      Posicionavel* ocupante[3];
      //ponteiro de duas posições de posicionavel (caso de peixe+ planta)
      // 0 é pedra
      // 1 é planta
      // 2 é peixe
} unidade;
```

```
//plano formado pelas unidades minimas
typedef struct {
      unidade** grid;
      // matriz [x][z] de variaveis local
} plano;

//cubo formado pelos planos
typedef struct {
      plano* dimensao;
      //vetor de [y] planos
} cubo;
```

Portanto cubo é um array de 3 dimensões onde cada unidade é um array de ponteiros para *Posicionavel*. Essa convenção usada pelo grupo facilitou na hora de decidir em métodos das superclasses (*Posicionavel* e *SerVivo*) qual é o tipo do objeto. Assim, bastava apenas o comparar retorno da função *Posicionavel::getId()* (0 para pedra, 1 para planta e 2 para peixe). Isto também aumentou bastante o nível de modularização do código, principalmente com relação aos métodos *aumentar( )* e *diminuir( )*, de *SerVivo*, que são chamados por mais de um metódo da classe *Peixe* e *Planta*. A classe *Ecossistema* pode ser vista a seguir.

```
// a ideia dessa classe é armazenar o "mapa" do ecossistema, localizações com ponteiros para os respectivos objetos
class Ecossistema
{
    private:
        static cubo aquario;
        static posicao limites;
    public:
        static posicao* getLimites();
        static void inicializar (int x, int y, int z);
        //inicializar tudo como nulo
        // arredores de pedra
        static Posicionavel** identificarOcupantes (int x, int y, int z);
        // retorna quem ocupa o local
        static void ocupar(int x,int y, int z, int i,Posicionavel* corpo);
        //seta posicionavel no cubo
};
```

A declaração de funções e atributos como estáticos permitiu que fossem acessados facilmente por qualquer parte do código. Por exemplo, caso alguma função de Peixe precisasse checar o cubo, bastaria chamar *Ecossistema::identificarOcupantes(x,y,z)*;

A classe *main* lê o arquivo descrito na especificação, faz as inicializações iniciais, chama a função *Ecossistema::inicializar( )* e define a localização de peixes, pedras e plantas. Também cria um objeto do tipo *Desenho*, e chama o loop *OpenGL*.

As funcionalidades de *Desenho* são descritas no próximo tópico.

4.2. OpenGL (Desenho.cpp, CCamera.cpp e carregadorObj.cpp)

A classe *Desenho* é classe principal do simulador, pois ela executa a lógica do simulador e trata da exibição gráfica.

O construtor de *Desenho* inicializa todos os valores necessários do *OpenGL*, definindo, por exemplo, a dimensão e posição da janela, a iluminação e o carregamento de texturas. O carregamento de texturas em especial é feito com auxílio da classe *carregadorOBJ*, de autoria do monitor da disciplina. As texturas dos objetos usadas foram adquiridas no site *TF3DM*, exceto a textura de pedra, que foi feita pelo grupo.

O método *display* da classe *Desenho* contêm a lógica do simulador, chamando o método *logica_simulador()*, e trata do desenho na tela.

A velocidade de execução de *display* é modificada usando o comando *Sleep* e a variável velocidade que é modificada utilizando as teclas de 1 a 5, variando a velocidade de 1 redesenho a cada segundo até 1 redesenho a cada milisegundo.

Em Desenho estão declaradas as funções que desenham água, pedra, peixe e planta.

Os códigos de *Desenho::desenhar_agua(), Desenho::desenhar_pedra()* e *Desenho::desenhar_planta()* são semelhantes. Todos desenham o seu elemento de forma simples. *Desenhar::planta()* utiliza uma escala de tamanho baseado na massa atual da planta.

A função *Desenho::desenhar_peixe()* é um pouco mais elaborada, permitindo rotação, escala de tamanho e variação de cor (com base na massa atual do peixe).

A rotação do peixe é feita em 26 ângulos possíveis (todo o cubo à sua volta) cujos angulos X, Y e Z foram definidos observando o objeto peixe enquanto alterava-se os angulos em 45º para observar se ele ficava numa posição natural ou não, esses ângulos foram inseridos numa tabela estática que é acessada no momento do desenho do peixe utilizando a direção atual para desenhar o peixe em uma das 26 direções possíveis.

A variação de cor de peixe se dá da cor normal (alaranjada) até a cor vermelha, quando o mesmo está prestes a explodir.

Para a câmera o grupo criou uma classe específica chamada *CCamera* com base num tutorial que encontramos no site *codecolony.de*. *CCamera* trabalha basicamente com soma de angulos para rotação da câmera e soma de vetores para os demais movimentos. A câmera é iniciada na origem com direção -z e com angulos relativos aos três eixos zero, para a movimentação da câmera em qualquer eixo é utilizada uma soma de vetores.

O principal problema que tivemos quanto a câmera foi na caminhada em três eixos simultâneos, que acontece após girar a câmera num angulo qualquer em dois eixos. Para a solução disto utilizamos um

atributo direção que indica com numeros de -1 a 1 nos três eixos que é obtida utilizando os ângulos de rotação no momento que é necessário saber a direção da câmera, a partir disto a distância da caminhada é multiplicada pelo vetor direção e somada ao vetor posição.

Para uma melhor visualização do aquário utilizamos a visão perspectiva. Para utilizá-la precisa-se dos métodos *glFrustrum()* e *gluPerspective()* que deformam o cubo de visualização de forma a simular diferença de distância deixando objetos mais distantes menores e os mais próximos maiores o que permite também uma melhor navegação pela parte interna do aquário.

O método *keyPressed* da classe *Desenho* mapeia teclas a movimentos da câmera:

- W - faz a câmera ir pra frente.
- S - faz a câmera ir para trás.
- A - gira a câmera em torno de seu eixo Y para a esquerda
- D - gira a câmera em torno de seu eixo Y para a direita
- Q - gira a câmera em torno de seu eixo X para frente
- E - gira a câmera em torno de seu eixo X para trás
- I - faz a câmera subir
- K - faz a câmera descer
- J - faz a câmera caminhar para o lado esquerdo
- L - faz a câmera caminhar para o lado esquerdo
- ESC - finaliza o programa
- 1 - redesenha a tela a cada 1 segundo
- 2 - redesenha a tela a cada 650 milisegundos
- 3 - redesenha a tela a cada 150 milisegundos
- 4 - redesenha a tela a cada 40 milisegundos
- 5 - redesenha a tela a cada 1 milisegundo

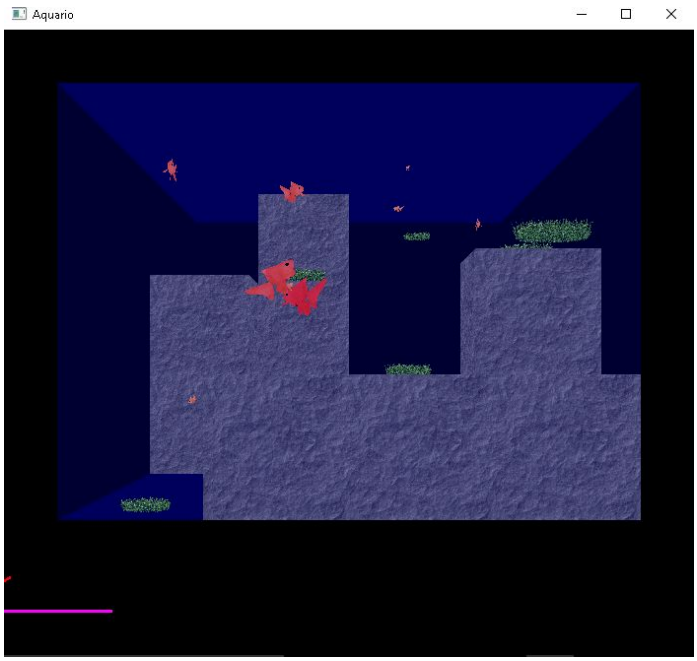
As instruções de compilação de nossa implementação estão descritas no arquivo *LEIAME.txt*, enviado em anexo a este relatório.

5. Execução do Projeto

Abaixo os prints com a entrada de acordo com o exemplo dado na especificação.

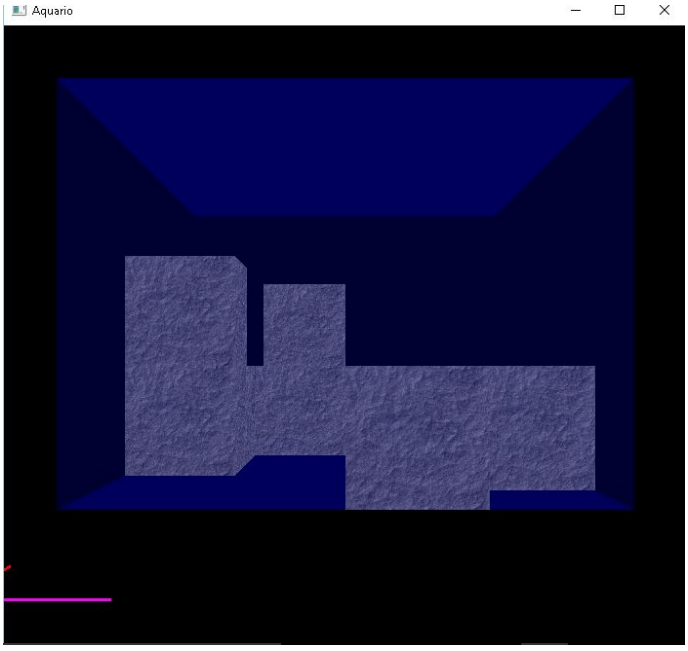
Planta dominando

Peixe prestes a explodir (cor vermelha)

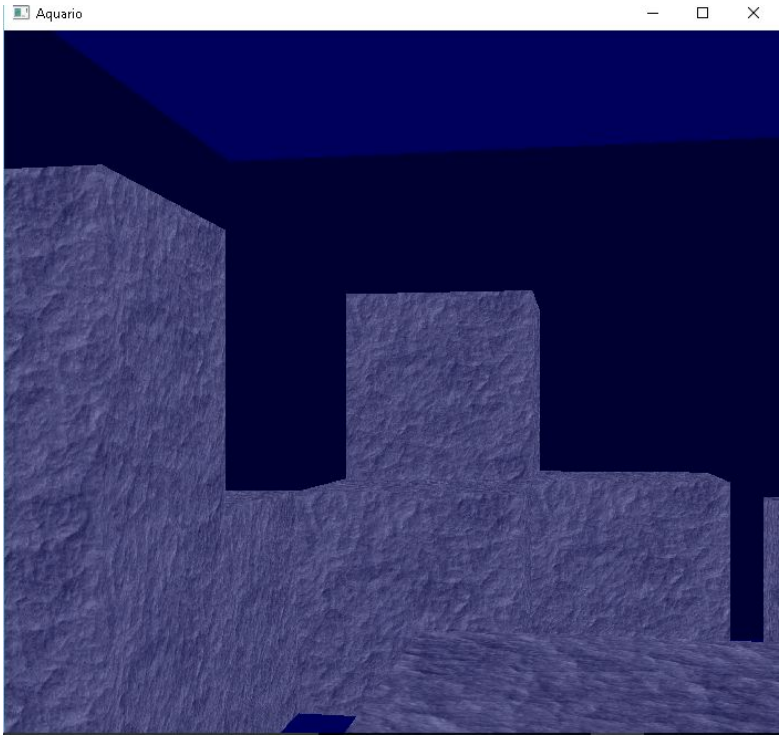


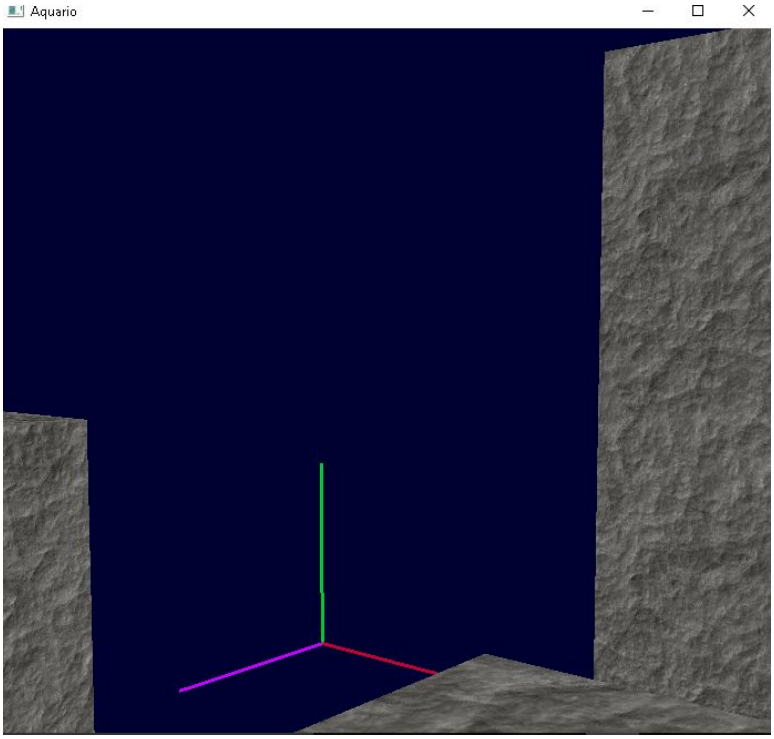
Ecossistema equilibrado

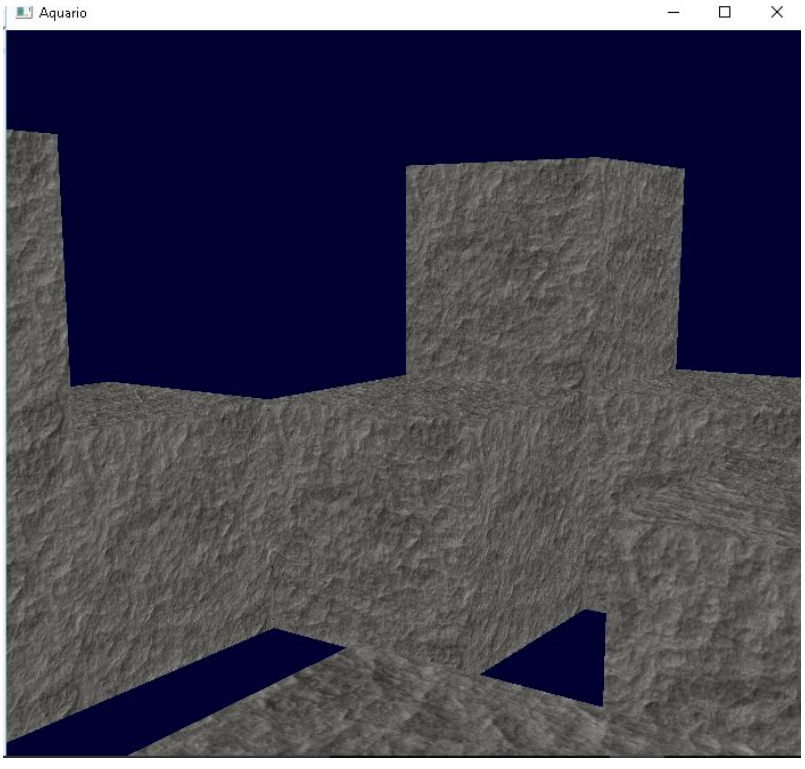
Peixe dominando



Morte total







Movimento da camera

6. Conclusão

Superando dificuldades tanto em nivel de linguagem quanto em nivel lógico, o grupo conseguiu implementar o projeto conforme as especificações (leitura, estruturas de dados, comportamento de objetos, desenho de objetos e matriz, camera, etc).

Depois de dias de trabalho é possivel afirmar que os conceitos ministrados na primeira unidade da disciplina foram melhor absorvidos.