

Relatório de Implementação - 2hu

Fábio de Azevedo Gomes - Cartão 00287696
Felipe de Almeida Graeff - Cartão 00261606

December 2019

1 Introdução

Este trabalho foi desenvolvido procurando se assemelhar a jogos do estilo *bullet hell*, nos quais o jogador deve ser capaz de desviar de uma quantidade potencialmente grande de projéteis que se movimentam pela tela. Estes tipos de jogos são geralmente baseados em uma *engine* bidimensional, porém pensamos em utilizar uma *engine* tridimensional para permitir controle maior sobre a câmera e sua posição.

2 Processo de desenvolvimento

Para a implementação deste jogo foi utilizado o código fonte do laboratório 4 resolvido pela dupla como base, pois o mesmo apresentava os controles básicos para movimentação da câmera livre e câmera *look-at*.

2.1 Câmera

Foram implementadas as duas formas de movimentação da câmera: *look-at* e livre, compondo duas formas de visualização:

- *First Person Perspective*
- *Third Person Perspective*

A câmera livre foi a primeira a ser implementada, sendo convertida para uma câmera "walk" através da projeção e normalização do vetor \overrightarrow{view} na superfície XZ global. Desta forma, quando o usuário pressiona uma tecla de movimento a câmera é movida ao longo destes eixos, independentemente se \overrightarrow{view} "aponta para cima" ou "para baixo"

A câmera *look-at* foi implementada da mesma maneira que a implementação do laboratório 4, com a pequena alteração de que o ponto sendo observado é o personagem que será controlado pelo usuário, e portanto a câmera se move junto ao movimentar o mesmo pela cena.

2.2 Objetos Fixos (imóveis)

Após implementadas as câmeras foi dado início o trabalho na implementação de objetos na cena. Utilizamos diversos modelos em formato *Wavefront* na aplicação, carregados através da biblioteca `tinyobjloader` e construídos na cena, como pode ser visto a seguir:

```
ObjModel floormodel ("../../data/plane.obj");
ComputeNormals(&floormodel);
BuildTrianglesAndAddToVirtualScene(&floormodel);

ObjModel mikumodel ("../../data/miku.obj");
ComputeNormals(&mikumodel);
BuildTrianglesAndAddToVirtualScene(&mikumodel);

ObjModel wallmodel ("../../data/wall.obj");
ComputeNormals(&wallmodel);
BuildTrianglesAndAddToVirtualScene(&wallmodel);

ObjModel enemy0model ("../../data/cow.obj");
ComputeNormals(&enemy0model);
BuildTrianglesAndAddToVirtualScene(&enemy0model);
```

Procuramos tornar o carregamento e renderização da cena o mais modular possível, de forma que possamos facilmente trocar entre cenas com diferentes objetos, e portanto desenvolvemos as seguintes estruturas e trechos de código:

```
struct PlacedObject {
    int id;                                //ID do objeto
    std::string name;                        //Nome do objeto
    glm::vec3 scale;                         //Escala do objeto
    glm::vec4 position_world;                //Posicao do objeto na cena
    glm::vec3 rotation;                      //Rotacao do objeto na cena
};

std::vector<PlacedObject> g_PlacedObjects;
```

Como a `struct SceneObject` guardava apenas informações sobre o modelo do objeto desenvolvemos uma segunda `struct`, `PlacedObject`, a qual guarda informações sobre uma cópia específica deste objeto. Temos também um vetor `g_PlacedObjects`, o qual é alterado toda vez que uma nova cena é carregada, mantendo em memória apenas as cópias dos objetos que estão presentes. Desta forma a renderização da cena se torna modular, e podemos realizar a mesma em apenas um trecho do código como é feito abaixo:

```

for (std :: vector<PlacedObject>::iterator it = g_PlacedObjects . begin
() ; it != g_PlacedObjects . end () ; it++)
{
    model = Matrix_Translate (it->position_world . x , it->
    position_world . y , it->position_world . z )
    * Matrix_Scale (it->scale . x , it->scale . y , it->scale . z )
    * Matrix_Rotate_Z (it->rotation . z )
    * Matrix_Rotate_Y (it->rotation . y )
    * Matrix_Rotate_X (it->rotation . x );
    glUniformMatrix4fv (model_uniform , 1 , GL_FALSE , glm :: value_ptr (
    model));
    glUniform1i (object_id_uniform , it->id );
    DrawVirtualObject (it->name . c _str ());
}

```

Notar que esta renderização faz uso das matrizes vistas em aula - Translação, Escala e Rotaão - Para posicionar o objeto na cena virtual.

2.2.1 Colisões com estes objetos

O vetor `g_PlacedObjects` também é utilizado para realizar testes de colisão com o ambiente, e devido à sua variabilidade dependente da cena podemos simplesmente realizar os testes de colisão em um só trecho do código, testando todos os seus componentes como é feito abaixo:

```

std :: vector<PlacedObject>::iterator it ;
for (it = g_PlacedObjects . begin () ; it != g_PlacedObjects . end () ; it++)
{
    if (hasBoxBoxCollision ( g_VirtualScene [ it->name ] ,
        it->position_world ,
        it->scale ,
        it->rotation ,
        mikuSceneObj ,
        player->position_world ,
        glm :: vec3 (1.0 f , 1.0 f , 1.0 f ) ,
        glm :: vec3 (0.0 f , g_CameraTheta , g_CameraPhi ))
    )
    {
        player->position_world = player->last_position_world ;
    }
}

```

Aqui `mikuSceneObj` corresponde ao modelo do personagem controlado pelo usuário, que só é renderizado quando se está utilizando *Third Person Perspective*.

Aqui também pode ser visto o primeiro teste de colisão implementado, teste entre duas *bounding boxes*, cujo propósito na aplicação é impedir que o usuário atravesse estes objetos fixos da cena.

```

bool hasBoxBoxCollision(SceneObject obj1, glm::vec4 pos_obj1, glm::
    vec3 scale_obj1, glm::vec3 rttm_obj1, SceneObject obj2, glm::
    vec4 pos_obj2, glm::vec3 scale_obj2, glm::vec3 rttm_obj2)
{
    if ((rttn_obj1.y >= -3.14/4 && rttm_obj1.y <= 3*3.14/4)
        || (rttn_obj1.y >= 3*-3.14/4 && rttm_obj1.y <= -3.14/4))
    {
        return ((obj1.bbox_min.z*scale_obj1.x) + pos_obj1.x <= (obj2.
            bbox_max.x*scale_obj2.x) + pos_obj2.x
            && (obj1.bbox_max.z*scale_obj1.x) + pos_obj1.x >= (obj2.
                bbox_min.x*scale_obj2.x) + pos_obj2.x)
            &&((obj1.bbox_min.y*scale_obj1.y) + pos_obj1.y <= (obj2.
                bbox_max.y*scale_obj2.y) + pos_obj2.y)
            && (obj1.bbox_max.y*scale_obj1.y) + pos_obj1.y >= (obj2.
                bbox_min.y*scale_obj2.y) + pos_obj2.y)
            &&((obj1.bbox_min.z*scale_obj1.z) + pos_obj1.z <= (obj2.
                bbox_max.z*scale_obj2.z) + pos_obj2.z)
            && (obj1.bbox_max.z*scale_obj1.z) + pos_obj1.z >= (obj2.
                bbox_min.z*scale_obj2.z) + pos_obj2.z);
    }
    else
    {
        return ((obj1.bbox_min.x*scale_obj1.x) + pos_obj1.x <= (
            obj2.bbox_max.x*scale_obj2.x) + pos_obj2.x
            && (obj1.bbox_max.x*scale_obj1.x) + pos_obj1.x >= (
                obj2.bbox_min.x*scale_obj2.x) + pos_obj2.x)
            &&((obj1.bbox_min.y*scale_obj1.y) + pos_obj1.y <= (
                obj2.bbox_max.y*scale_obj2.y) + pos_obj2.y)
            && (obj1.bbox_max.y*scale_obj1.y) + pos_obj1.y >= (
                obj2.bbox_min.y*scale_obj2.y) + pos_obj2.y)
            &&((obj1.bbox_min.z*scale_obj1.z) + pos_obj1.z <= (
                obj2.bbox_max.z*scale_obj2.z) + pos_obj2.z)
            && (obj1.bbox_max.z*scale_obj1.z) + pos_obj1.z >= (
                obj2.bbox_min.z*scale_obj2.z) + pos_obj2.z);
    }
}

```

Em suma, é verificado se algum componente da *bounding box* do objeto 1 está contido na *boudning box* do objeto 2, feito em coordenadas globais e escalados corretamente. Como os objetos fixos da cena estão apenas rotacionados em múltiplos de 90° o teste foi simplificado, apenas invertendo os componentes **x** e **z** das **bounding boxes** dos objetos.

2.3 Iluminação e Texturas

Foram utilizadas diversas texturas na aplicação, carregadas através da função `LoadTextureImage` utilizada no laboratório 5. Estas texturas são carregadas e enviadas para o *shader* de fragmentos, o qual computa, para cada ponto do objeto, a posição U,V correta da textura a ser mapeada, além de computar a equação de iluminação apropriada para o objeto, conforme apresentado:

```

if ( object_id == MENU.BACKGROUND )
{
    vec4 v_lin = (position_world - camera_position);
    v_lin = v_lin/norm(v_lin);

    vec4 p_lin = camera_position + v_lin;
    vec4 coord_vector = (p_lin - camera_position);
    float theta = atan(coord_vector.x,coord_vector.z);
    float phi = asin(coord_vector.y);

    U = (theta + M_PI)/(2*M_PI);
    V = (phi + M_PI_2)/M_PI;

    Kd = texture(TextureImage1, vec2(U,V)).rgb;

    Ks = vec3(0.0,0.0,0.0);
    Ka = vec3(0.0,0.0,0.0);
    q = 1.0;

    // Invertemos a normal da esfera, para que ela esteja "virada para dentro"
    n = -n;
}

```

Aqui temos o mapeamento da textura de fundo da aplicação, para o caso do menu. A normal é invertida neste caso para que a esfera esteja "virada para dentro", de forma a não computar equações de iluminação de forma incorreta. As equações de iluminação são computadas de acordo com a fórmula do modelo de *Blinn-Phong*, segundo os seguintes trechos presentes no *Fragment Shader*:

```

// vetor h intermediário entre l e v
vec4 h = (l + v)/length(l+v);

vec3 lambert_diffuse_term = Kd*I*max(0,dot(n,l));
vec3 ambient_term = Ka*Ia;
vec3 blinn_phong_specular_term = Ks*I*pow(max(0,dot(n,h)),q);

color = lambert_diffuse_term + ambient_term + blinn_phong_specular_term;

```

Desta forma somos capazes de produzir tanto objetos com iluminação difusa quanto objetos com iluminação espelhada pelo modelo de iluminação *Blinn-Phong*, bastando decidir o coeficiente espelhado **Ks** do objeto como sendo nulo ou não.

2.4 Lógica de jogo

Dados um sistema de câmeras e um sistema de posicionamento de objetos foi iniciado o desenvolvimento da aplicação propriamente dita. Foram criadas 4 **structs** para uso na aplicação, as quais representam objetos móveis do jogo que podem interagir entre si. Estes objetos requerem processamento diferenciado daqueles fixos em caso de colisão, e portanto são mantidos em variáveis separadas.

```

struct Player {
    float health_points;
    glm::vec4 last_position_world;
    glm::vec4 position_world;
    glm::vec4 velocity;
    float movement_speed;

    Player();
    void compute_movement(double delta_t);
    void reset_stats();
    void process_movement_input(glm::vec4 camera_view_vector);

};


```

A **struct Player** representa o personagem sendo controlado pelo jogador, contendo informações como sua vida, posição, vetor velocidade e velocidade de movimento. O *input* do usuário é obtido através da função **process_movement_input**, a qual atualiza o vetor velocidade do personagem de acordo com a posição para qual a câmera estava olhando naquele momento. Este vetor velocidade é posteriormente utilizado pela função **compute_movement**, em conjunto com uma variação de tempo ΔT , que efetivamente move o personagem do jogador. A função **reset_stats** apenas se encarrega de resetar a vida, posição e velocidade do jogador em caso de troca de cena.

```

struct EnemyObject {
    int id;
    float health_points;
    int attack_patterns[3];
    int attack_cycle;
    PlacedObject* body;
    float bezier_t;
    float movement_speed;
    int texture;

    EnemyObject (int enemyId, PlacedObject* enemyBody);
    void attack (int pattern, Player* player);
    void compute_movement(double delta_t, float arena_size);
};


```

A **struct EnemyObject** por sua vez representa o inimigo do jogo, presente nos dois níveis disponíveis. Sua movimentação é dada por uma curva de beziér de grau 3 (4 pontos de controle) conforme a fórmula estudada na disciplina

$$c(t) = (1-t)^3 * p_1 + 3t(1-t)^2 * p_2 + 3t^2(1-t) * p_3 + t^3 * p_4$$

computada para as três coordenadas x, y e z e utilizada para obter a nova posição do inimigo. A velocidade com a qual o parâmetro t varia depende do intervalo de tempo ΔT informado e da velocidade de movimentação deste inimigo.

A função `attack` é utilizada para gerar o próximo objeto do jogo: *Bullets*, e pode ocorrer em diversos padrões pré-definidos, descritos pelas constantes

```
#define ATK_EXPANDING_CIRCLE 0
#define ATK_RANDOM_SPREAD 1
#define ATK_DIRECTED_CONE 2
#define ATK_EXPANDING_SPIRAL 3
#define ATK_CLOSING_CIRCLE 4
#define ATK_BREAKING_CIRCLE 5
```

A figura 1 apresenta um exemplo do ataque descrito por `ATK_DIRECTED_CONE` sendo utilizado pela vaca:

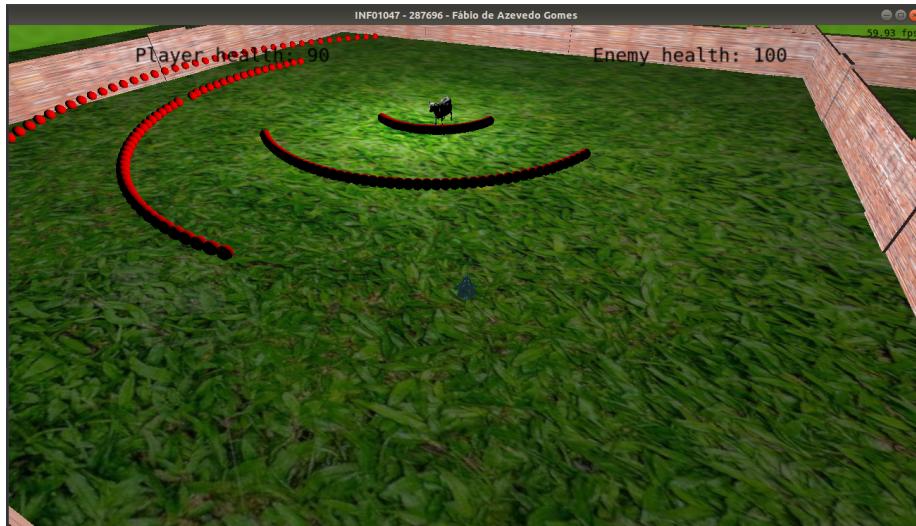


Figure 1: Vaca utilizando o ataque `ATK_DIRECTED_CONE`, gerando 50 *bullets* em um cone direcionado ao jogador

```
struct BulletObject {
    int id;
    glm::vec4 position_world;
    glm::vec4 color;
    float damage;
    float bullet_t;
    glm::vec4 velocity;
};

std::vector<struct BulletObject> g_BulletsOnPlay;
```

A `struct BulletObject` representa um projétil que é lançado pelo inimigo, denominado *bullet*, que pode ser visto também na figura 1. Utilizamos um vetor separado, `g_BulletsOnPlay` para representar as *bullets* ativas, visto que

ao colidir com o jogador o processamento feito é diferente dos objetos fixos da cena. O parametro `bullet.t` é utilizado para algumas *bullets* especiais que não se movem imediatamente, apenas após algum tempo.

A cada iteração, são feitos dois testes de colisão para cada *bullet* ativa: Se colidiu com a parede ou se colidiu com o jogador.

```
for (std::vector<PlacedObject>::iterator obj_it = g_PlacedObjects.begin(); !collided && obj_it != g_PlacedObjects.end(); obj_it++)
{
    if (hasPointBoxCollision(it->position_world,
                             g_VirtualScene[obj_it->name],
                             obj_it->position_world,
                             obj_it->scale,
                             obj_it->rotation))
    {
        g_BulletsOnPlay.erase(it);
        collided = true;
    }
}
```

Primeiro é feito um teste de colisão com os objetos fixos da cena, onde é utilizado um teste de intersecção ponto-cubo, implementado de forma simples, bastando verificar se o ponto pertence à *bounding box* do objeto:

```
bool hasPointBoxCollision(glm::vec4 point, SceneObject obj, glm::vec4 pos_obj, glm::vec3 scale_obj, glm::vec3 rttm_obj)
{
    if ((rttm_obj.y >= -3.14/4 && rttm_obj.y <= 3*3.14/4)
        || (rttm_obj.y >= 3*-3.14/4 && rttm_obj.y <= -3.14/4))
    {
        return (point.x > (obj.bbox_min.z*scale_obj.x) + pos_obj.x
                && point.x < (obj.bbox_max.z*scale_obj.x) + pos_obj.x)
               && (point.y > (obj.bbox_min.y*scale_obj.y) + pos_obj.y
                    && point.y < (obj.bbox_max.y*scale_obj.y) + pos_obj.y)
               && (point.z > (obj.bbox_min.x*scale_obj.z) + pos_obj.z
                    && point.z < (obj.bbox_max.x*scale_obj.z) + pos_obj.z);
    }
    else
    {
        return (point.x > (obj.bbox_min.x*scale_obj.x) + pos_obj.x
                && point.x < (obj.bbox_max.x*scale_obj.x) + pos_obj.x)
               && (point.y > (obj.bbox_min.y*scale_obj.y) + pos_obj.y
                    && point.y < (obj.bbox_max.y*scale_obj.y) + pos_obj.y)
               && (point.z > (obj.bbox_min.z*scale_obj.z) + pos_obj.z
                    && point.z < (obj.bbox_max.z*scale_obj.z) + pos_obj.z);
    }
}
```

Logo após é feito um teste de colisão com o jogador, que utiliza o teste de intersecção esfera-cubo seguinte:

```

bool hasSphereBoxCollision(glm::vec4 sphere_center, float
    sphere_radius, SceneObject obj, glm::vec4 pos_obj, glm::vec3
    scale_obj, glm::vec3 rttm_obj)
{
    float r_sq = sphere_radius*sphere_radius;
    float d_min = 0;

    if ((rttm_obj.y >= -3.14/4 && rttm_obj.y <= 3*3.14/4)
        || (rttm_obj.y >= 3*-3.14/4 && rttm_obj.y <= -3.14/4))
    {
        //X
        if (sphere_center.x < (obj.bbox_min.z*scale_obj.x) +
            pos_obj.x)
            d_min += sqrt(sphere_center.x - obj.bbox_min.z);
        else if (sphere_center.x > (obj.bbox_max.z*scale_obj.x) +
            pos_obj.x)
            d_min += sqrt( sphere_center.x - obj.bbox_max.z);
        //Y
        if (sphere_center.y < (obj.bbox_min.y*scale_obj.y) +
            pos_obj.y)
            d_min += sqrt( sphere_center.y - obj.bbox_min.y);
        else if (sphere_center.y > (obj.bbox_max.y*scale_obj.y) +
            pos_obj.y)
            d_min += sqrt( sphere_center.y - obj.bbox_max.y);
        //Z
        if (sphere_center.z < (obj.bbox_min.x*scale_obj.z) +
            pos_obj.z)
            d_min += sqrt( sphere_center.z - obj.bbox_min.x);
        else if (sphere_center.z > (obj.bbox_max.x*scale_obj.z) +
            pos_obj.z)
            d_min += sqrt( sphere_center.z - obj.bbox_max.x);
    }
    else
    {
        for (int i =0; i < 3; i++)
        {
            if (sphere_center[i] < (obj.bbox_min[i]*scale_obj[i]) +
                pos_obj[i])
                d_min += sqrt( sphere_center[i] - obj.bbox_min[i]);
            else if (sphere_center[i] > (obj.bbox_max[i]*scale_obj[i]) +
                pos_obj[i])
                d_min += sqrt( sphere_center[i] - obj.bbox_max[i]);
        }
    }
    return d_min <= r_sq;
}

```

Estas *bullets* ao atingirem a parede, o jogador ou se movimentarem para muito longe do centro da cena são descartadas, para questões de eficiência de uso de memória em tempo de execução.

```

struct ProjectileObject {
    int id;
    float damage;

```

```

    glm::vec4 velocity;
    PlacedObject* model;
    float traveled_distance;
};

std::vector<struct ProjectileObject> g_ProjectilesOnPlay;

```

Por fim, temos a **struct ProjectileObject**, a qual mantém informações sobre os projéteis lançados pelo jogador. Similar às *bullets*, estes projéteis se movimentam com base em um intervalo de tempo ΔT , porém têm uma distância máxima que podem viajar. A cada iteração são feitos testes de intersecção ponto-cubo para cada projétil presente em **g_ProjectilesOnPlay** para verificar se atinge o inimigo, e também são feitos testes para testar se o projétil atingiu sua distância máxima de movimentação.

3 Manual

Para jogar utilize as teclas W,A,S,D para movimentação, move a câmera com o mouse e aperte o botão esquerdo do mouse para atirar seus projéteis. Pressionar a tecla ESC encerra o jogo.

Existem duas portas na cena inicial, as quais levam para os dois níveis disponíveis: A fazenda e a fábrica. Para acessar o nível basta se aproximar da porta, e serão dados alguns *frames* de invencibilidade ao entrar no jogo. Cada porta possui um marcador que representa se aquele nível foi vencido ou não, conforme a figura 2.



Figure 2: Portas para seleção do nível, junto de seus marcadores

Dentro de um nível, haverá um inimigo se movendo pela arena, o qual deve ser derrotado. Estes inimigos lançam diversas *bullets* que devem ser desviadas



Figure 3: Medalha disponibilizada ao derrotar o inimigo

para evitar que o jogador morra. Dependendo do inimigo, a forma como suas *bullets* são lançadas varia, então não é possível utilizar a mesma estratégia para vencer ambos.

Ao derrotar o inimigo, as *bullets* e os projéteis serão limpados da tela, restando apenas coletar a medalha que será disponibilizada no centro da arena conforme a figura 3. A coleta desta medalha levará o jogador devolta à cena inicial, na qual pode escolher o nível novamente.



Figure 4: Nível: Fazenda

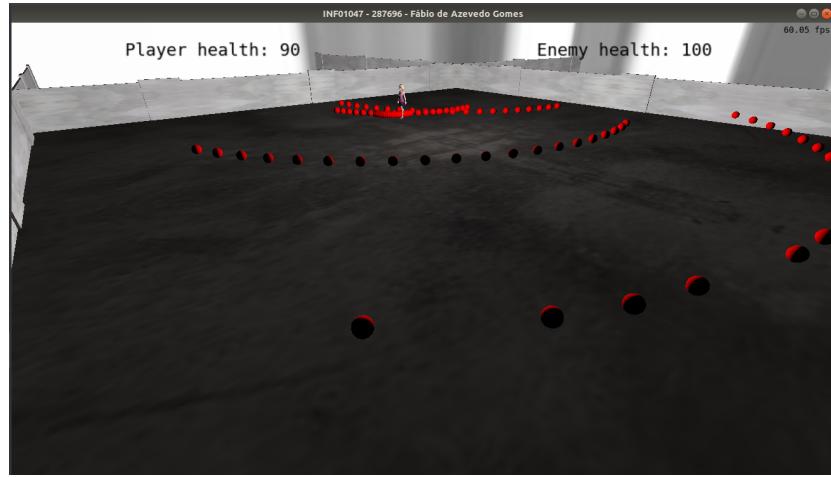


Figure 5: Nível: Fábrica

4 Compilação e execução

O programa pode ser compilado através da IDE CodeBlocks, bastando carregar o projeto e utilizar a opção "build". O projeto também pode ser compilado utilizando o `makefile` disponível junto na pasta do projeto. Neste caso o programa deve ser executado do diretório `bin/Linux/` ou `bin/Release/` para que possa encontrar as dependências em tempo de execução.