

# Trabalho Final

Fábio de A. Gomes<sup>1</sup>    Felipe de Almeida Graeff<sup>1</sup>

<sup>1</sup>Universidade Federal do Rio Grande do Sul

INF01047 - Fundamentos de Computação Gráfica, 2019

# Ideia Geral



- ▶ O Objetivo do desenvolvimento foi tentar implementar um jogo no estilo *bullet-hell* inspirado em jogos como *Touhou*.
- ▶ O jogador deve desviar de uma quantidade grande de projéteis que se movem de maneira definida.
- ▶ Estes projéteis geralmente seguem algum padrão (Eg. Círculo, Espiral, etc.)

## Base da Implementação

- ▶ Foi utilizado o código fonte do laboratório 4 resolvido pelos integrantes do grupo como base
- ▶ Esta escolha foi devido ao fato de este laboratório conter a implementação das câmeras virtuais
- ▶ Esta escolha guiou a ordem do processo de implementação

# Ordem de Implementação

- ▶ Procuramos dividir o processo em partes que tornariam o desenvolvimento incremental:
  - ▶ Câmera e Movimentação
  - ▶ Criação de Cenas
  - ▶ Iluminação
  - ▶ Colisões
  - ▶ Lógica de jogo
  - ▶ Conteúdo

# Câmera e Movimentação

```
if (!firstPersonView)
{
    float r = g_CameraDistance;
    float y = r*sin(g_CameraPhi);                                + player->position_world.y;
    float z = r*cos(g_CameraPhi)*cos(g_CameraTheta) + player->position_world.z;
    float x = r*cos(g_CameraPhi)*sin(g_CameraTheta) + player->position_world.x;
    camera_position_c = glm::vec4(x,y,z,1.0f);
    camera_lookat_l = player->position_world;
    camera_view_vector = camera_lookat_l - camera_position_c;
}
else
{
    camera_position_c = player->position_world;
    camera_view_vector =
        (Matrix_Rotate_Y(g_CameraTheta)*Matrix_Rotate_X(-g_CameraPhi))
        *glm::vec4(0.0f,0.0f,-1.0f,0.0f);
}
```

Figure: Definição das duas câmeras virtuais implementadas

# Câmera e Movimentação

```
void process_movement_input(glm::vec4 camera_view_vector)
{
    glm::vec4 up_vector = glm::vec4(0.0f, 1.0f, 0.0f, 0.0f);

    glm::vec4 view_projection = (camera_view_vector * glm::vec4(1.0f, 0.0f, 1.0, 0.0f));
    view_projection = view_projection / norm(view_projection);

    glm::vec4 side_vector = crossproduct(up_vector, -view_projection);
    side_vector = side_vector / norm(side_vector);

    velocity = glm::vec4(0.0f, 0.0f, 0.0f, 0.0f);

    if (pressed[GLFW_KEY_W]) velocity += view_projection;
    if (pressed[GLFW_KEY_A]) velocity -= side_vector;
    if (pressed[GLFW_KEY_S]) velocity -= view_projection;
    if (pressed[GLFW_KEY_D]) velocity += side_vector;

    velocity = norm(velocity) == 0 ? velocity / norm(velocity) : velocity;
}
```

Figure: Método Player::process\_movement\_input

# Câmera e Movimentação

```
void compute_movement(double delta_t)
{
    last_position_world = position_world;
    position_world =
        position_world + velocity*(float)delta_t*(float)movement_speed;
}
```

Figure: Método Player::compute\_movement

- ▶ Aqui temos a movimentação propriamente dita do personagem pela cena

# Criação de Cenas

- ▶ Procuramos modularizar o processo de população de uma cena com objetos fixos
- ▶ Criamos uma estrutura de dados que guarda informações sobre as transformações sobre um objeto virtual
- ▶ Criamos um vetor que armazena estas estruturas.

## Criação de Cenas

```
struct PlacedObject {  
  
    int id;  
    std::string name;  
    glm::vec3 scale;  
    glm::vec4 position_world;  
    glm::vec3 rotation;  
};  
  
std::vector<PlacedObject> g_PlacedObjects;
```

Figure: Representação de objetos fixos na cena

- ▶ Desta forma conseguimos simplificar a renderização e testes de colisão com estes objetos

```
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());
}
```

Figure: Renderizando objetos fixos da cena

```
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(1.0f,1.0f,1.0f),
                           glm::vec3(0.0f, g_CameraTheta, g_CameraPhi)))
    ) {
        player->position_world = player->last_position_world;
    }
}
```

Figure: Testando por colisões com objetos fixos da cena

# Illuminação

- ▶ Implementamos o modelo de iluminação de Blinn-Phong, que permite tanto a simulação de objetos especulares quanto difusos
- ▶ Também utilizamos texturas para alguns objetos (eg. Vaca, Porta) através de mapeamentos em projeção planar e esférica
- ▶ Através de um grande *if* decidimos qual textura aplicar a qual objeto

## Illuminação

```
if ( object_id == MENU_BACKGROUND )  
{  
else if ( object_id == GRASS_FLOOR )  
{  
else if ( object_id == MIKU)  
{  
else if ( object_id == WALL)  
{  
else if ( object_id == ENEMY_COW)  
{  
else if ( object_id == BULLET)  
{  
else if ( object_id == PROJECTILE)  
{
```

Figure: Seleção de valores de coeficientes para cada objeto

# Illuminação

```
vec4 bbox_mid = (bbox_max + bbox_min) / 2;

vec4 p_lin = bbox_mid + ((position_model - bbox_mid)/length(position_model - bbox_mid));
vec4 coord_vector = (p_lin - bbox_mid);
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(TextureImage3, vec2(U,V)).rgb;

Ks = vec3(0.0,0.0,0.0);
Ka = vec3(0.0,0.0,0.0);
q = 20.0;
```

Figure: Mapeamento de texturas utilizando a projeção esférica

# Iluminação

```
vec3 lambert_diffuse_term = Kd*I*max(0,dot(n,l));  
  
vec3 ambient_term = Ka*Ia;  
  
vec4 h = l + v;  
h = h/length(h);  
  
vec3 blinn_phong_specular_term = Ks*I*pow(max(0,dot(n,h)),q);  
  
color = lambert_diffuse_term + ambient_term + blinn_phong_specular_term;
```

Figure: Equação de iluminação no modelo de Blinn-Phong

# Colisões

- ▶ Após termos uma cena com objetos, tentamos impedir que o jogador atravesse tais objetos.
- ▶ Todos os objetos da nossa cena estavam rotacionados em passos de 90°
- ▶ Foram realizados então testes de colisão utilizando uma aproximação da rotação dos mesmos

## Colisões

```
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)
```

Figure: Parte do teste de colisão caixa-caixa

# Colisões

```
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);
```

Figure: Teste de colisão ponto-caixa

# Colisões

```
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]);
```

Figure: Parte do teste de colisão círculo-caixa

# Lógica de Jogo

- ▶ O jogo desenvolvido possui 4 estruturas principais:
  - ▶ Jogador (Player)
  - ▶ Inimigo (EnemyObject)
  - ▶ Bullets (BulletObject)
  - ▶ Projetéis (ProjectileObject)

## Lógica de Jogo - Jogador

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

Figure: Estrutura do jogador

## Lógica de Jogo - Inimigo

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

Figure: Estrutura do inimigo

# Lógica de Jogo - Inimigo

```
#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

case ATK_EXPANDING_CIRCLE:
    for (int i = 0; i < 20; i++)
    {
        bullet = ((int)g_BulletsOnPlay.size(),
                   glm::vec4(body->position_world.x,
                             body->position_world.y,
                             body->position_world.z,
                             0.0f),
                   glm::vec4(1.0f, 0.0f, 0.0f, 0.0f),
                   10.0f,
                   0.0f,
                   Matrix_Rotate_Y(angle*i)*vel);

        g_BulletsOnPlay.push_back(bullet);
    }
    break;
```

## Lógica de Jogo - Bullets

```
struct BulletObject {  
  
    int id;                      /*Identificação ú  
    glm::vec4 position_world;    /*Posição desta bu  
    glm::vec4 color;            /*Cor desta bullet  
    float damage;               /*Dano que esta bu  
    float bullet_t;              /*t caso ela este  
    glm::vec4 velocity;          /*Velocidade da bu  
};  
  
std::vector<struct BulletObject> g_BulletsOnPlay;
```

Figure: Estrutura das bullets

## Lógica de Jogo - Projéteis

```
struct ProjectileObject {  
  
    int id;  
    float damage;  
    glm::vec4 velocity;  
    PlacedObject* model;  
    float traveled_distance;  
};  
  
std::vector<struct ProjectileObject> g_ProjectilesOnPlay;
```

Figure: Estrutura dos projéteis lançados pelo jogador

# Conteúdo

- ▶ Por fim, pudemos tirar proveito do fato de que toda a primeira cena foi criada de forma facilmente modularizável
- ▶ Criamos mais uma cena, com outro inimigo utilizando um modelo e conjunto de ataques diferente através da simples adição de uma nova cena com objetos mapeados com texturas diferentes.

# Conteúdo



Figure: Jogador lutando contra a vaca no nível fazenda

# Conteúdo

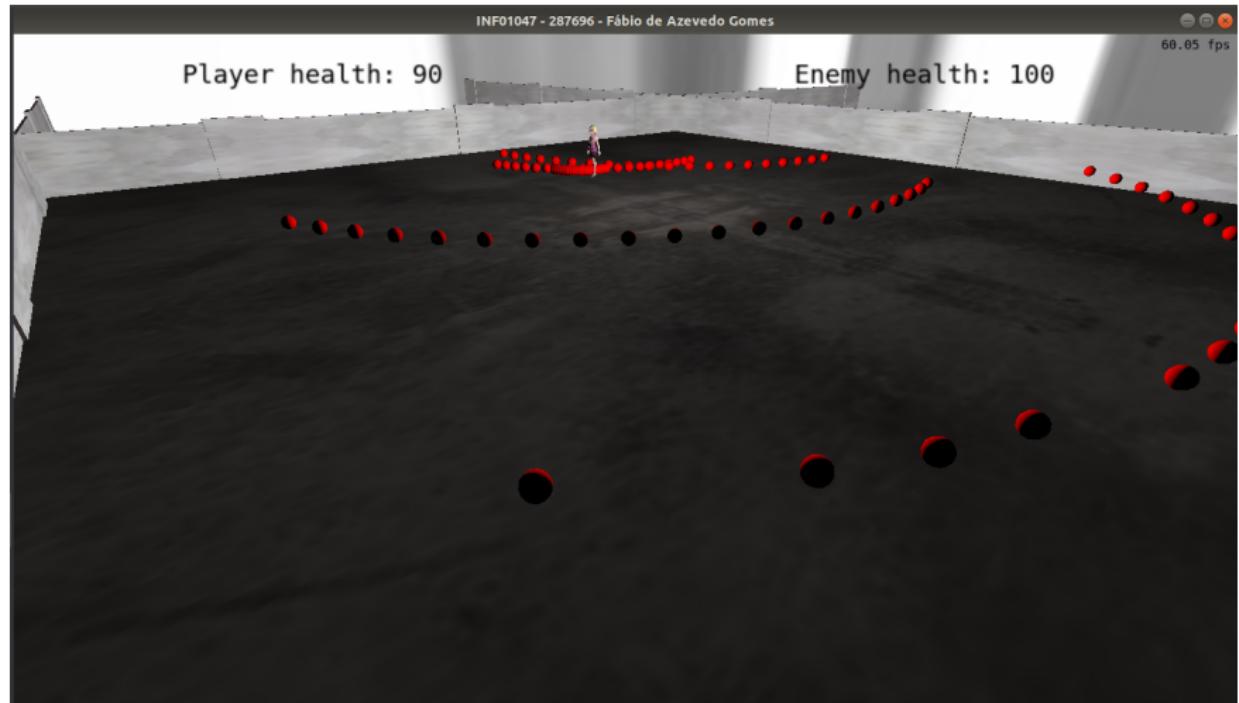


Figure: Jogador lutando contra o vampiro no nível fábrica