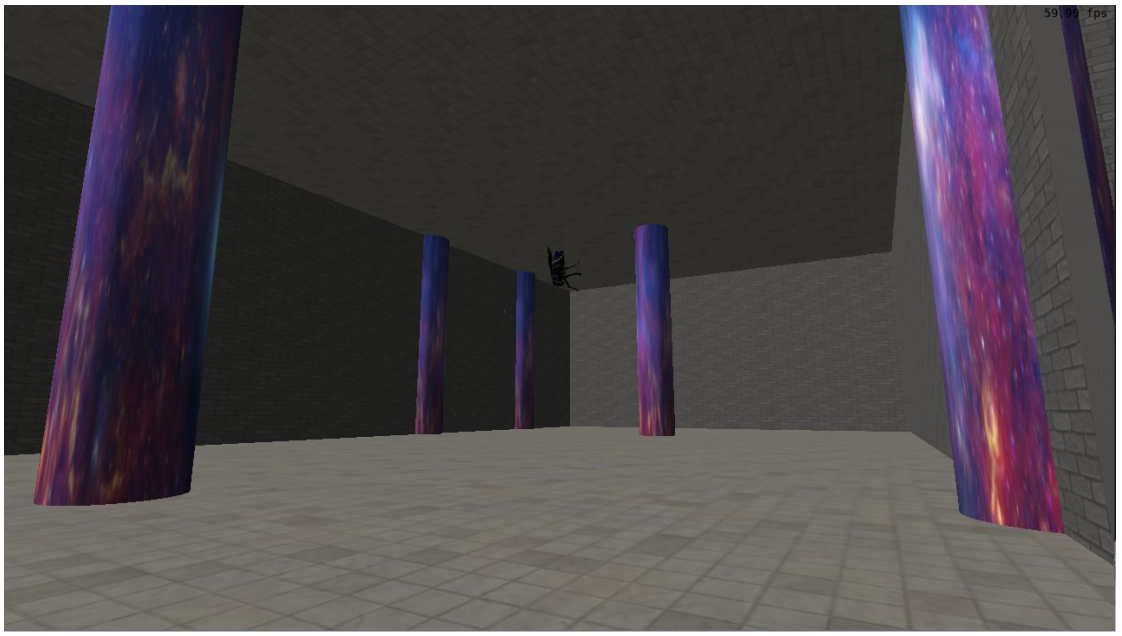
Fundamentos de computação gráfica
2020/1

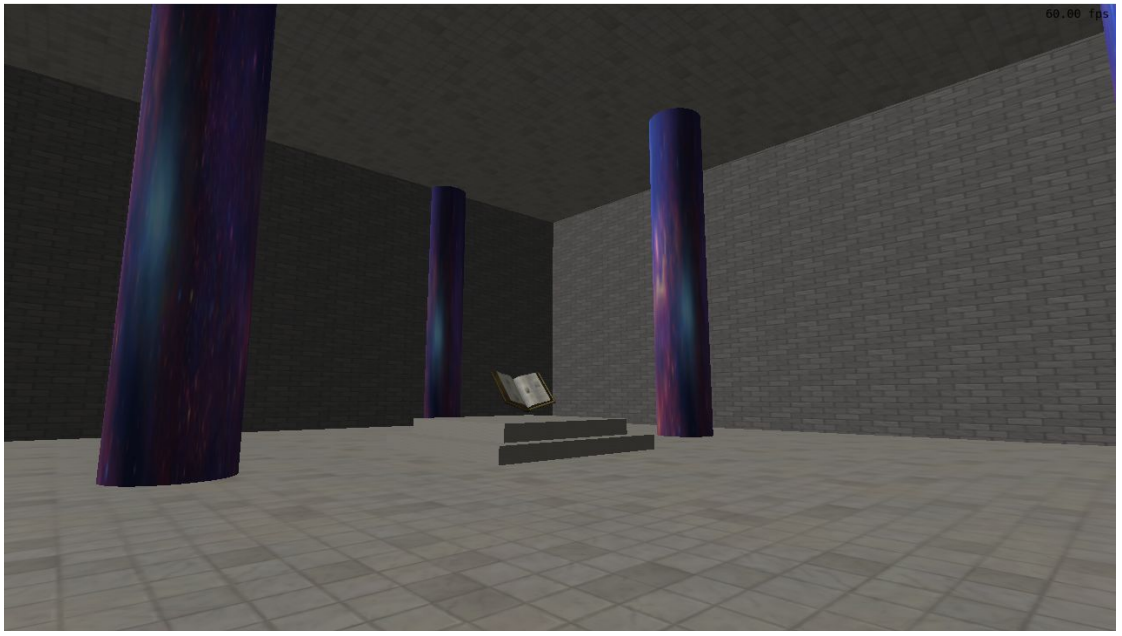
Implementação de um jogo em OpenGL

UFRGS
UNIVERSIDADE FEDERAL
DO RIO GRANDE DO SUL

.inf
INSTITUTO
DE INFORMÁTICA

Mateus Luiz Salvi
Rafael Oleques Nunes


59.99 fps


60.00 fps


60.00 fps

Como jogar

Os controles seguem o padrão de jogos em primeira pessoa, a sua movimentação é feita usando-se as teclas **W**, **A**, **S** e **D** e a visão é movimentada com o **mouse**.

Ao iniciar o jogo você será apresentado com uma mensagem explicando seu objetivo: encontrar um livro de magia, para que possa escapar do castelo.

Durante o jogo haverá um inimigo, que podem ser derrotado transformando-o em pedra; para isso, mire com o **centro da tela** em um deles, clique o **botão esquerdo do mouse**. Você pode empurrar o inimigo petrificado pelo cenário com o botão **esquerdo do mouse.**

Chegando ao livro o objetivo está concluído e o jogo termina. Você pode sair do jogo a hora que quiser pressionando a tecla **ESC**.

Não há uma maneira de falhar no jogo, então tome seu tempo.

Compilando o jogo:

É possível executar o jogo acessando a pasta bin/release e abrindo o arquivo wizard.exe.

O projeto do code:blocks é disponibilizado também, junto com todas as bibliotecas usadas, se necessário, pode-se compilar novamente através do code:blocks.

Windows

Para compilar e executar este projeto no Windows, baixe a IDE Code::Blocks em **http://codeblocks.org/** e abra o arquivo "Wizard".

ATENÇÃO: os "Build targets" padrões (Debug e Release) estão configurados para Code::Blocks versão 20.03 ou superior, que utiliza MinGW 64-bits. Se você estiver utilizando versões mais antigas do Code::Blocks (17.12 ou anteriores) você precisa alterar o "Build target" para "Debug (CBlocks 17.12 32-bit)" ou "Release (CBlocks 17.12 32-bit)" antes de compilar o projeto.

Linux

Para compilar e executar este projeto no Linux, primeiro você precisa instalar as bibliotecas necessárias. Para tanto, execute o comando abaixo em um terminal.

Esse é normalmente suficiente em uma instalação de Linux Ubuntu:

***sudo apt-get install build-essential make libx11-dev libxrandr-dev ***
***libxinerama-dev libxcursor-dev libxcb1-dev libxext-dev ***
libxrender-dev libxfixes-dev libxau-dev libxdmcp-dev

Se você usa Linux Mint, talvez seja necessário instalar mais algumas bibliotecas:

sudo apt-get install libmesa-dev libxxf86vm-dev

Após a instalação das bibliotecas acima, você possui duas opções para compilação: utilizar Code::Blocks ou Makefile.

Linux com Code::Blocks

Instale a IDE Code::Blocks (versão Linux em http://codeblocks.org/), abra o arquivo "Wizard.cbp", e modifique o "Build target" de "Debug" para "Linux".

Linux com Makefile

Abra um terminal, navegue até a pasta "Wizard", e execute o comando "make" para compilar. Para executar o código compilado, execute o comando "make run".

macOS

Para compilar e executar esse projeto no macOS, primeiro você precisa instalar o HOMEBREW, um gerenciador de pacotes para facilitar a instação de bibliotecas. O HOMEBREW pode ser instalado com o seguinte comando no terminal:

/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"

Após a instalação do HOMEBREW, a biblioteca GLFW deve ser instalada. Isso pode ser feito pelo terminal com o comando:

brew install glfw

macOS com Makefile

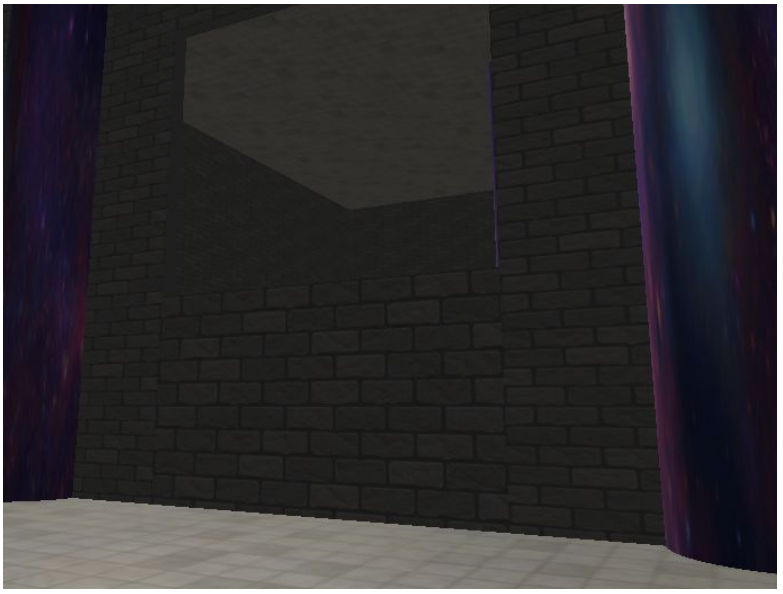
Abra um terminal, navegue até a pasta "Wizard", e execute o comando **"make -f Makefile.macOS"** para compilar. Para executar o código compilado, execute o comando **"make -f Makefile.macOS run"**.

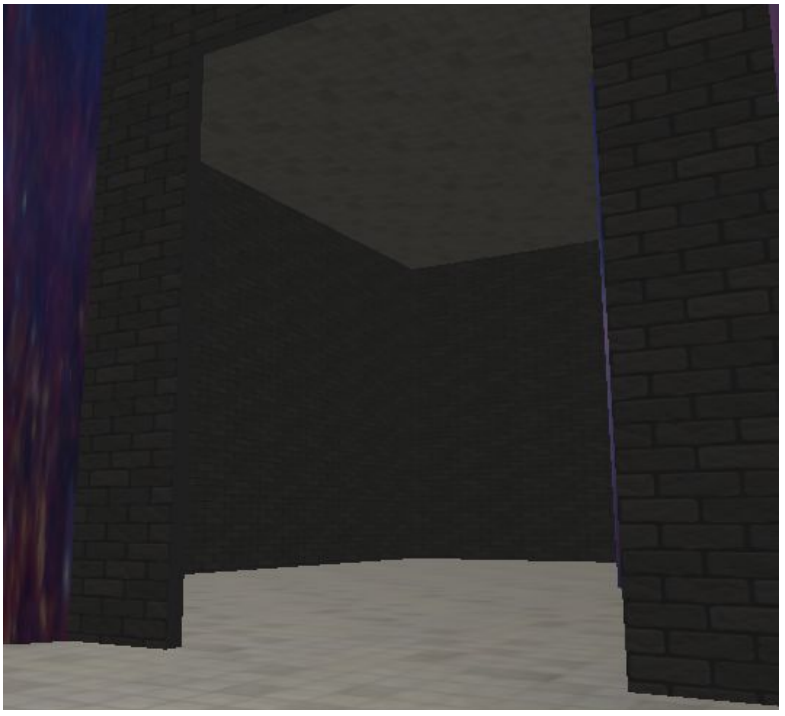
Observação: a versão atual da IDE Code::Blocks é bastante desatualizada para o macOS. A nota oficial dos desenvolvedores é: "Code::Blocks 17.12 for Mac is currently not available due to the lack of Mac developers, or developers that own a Mac. We could use an extra Mac developer (or two) to work on Mac compatibility issues.

Implementações

Animações baseadas no tempo:

Como exemplo, umas das animações baseadas no tempo é a da porta da primeira sala do jogo, que abre movendo-se para baixo. A movimentação dos inimigos e da câmera também são baseadas no tempo decorrido. Assim como o texto de objetivo apresentado ao iniciar o jogo.

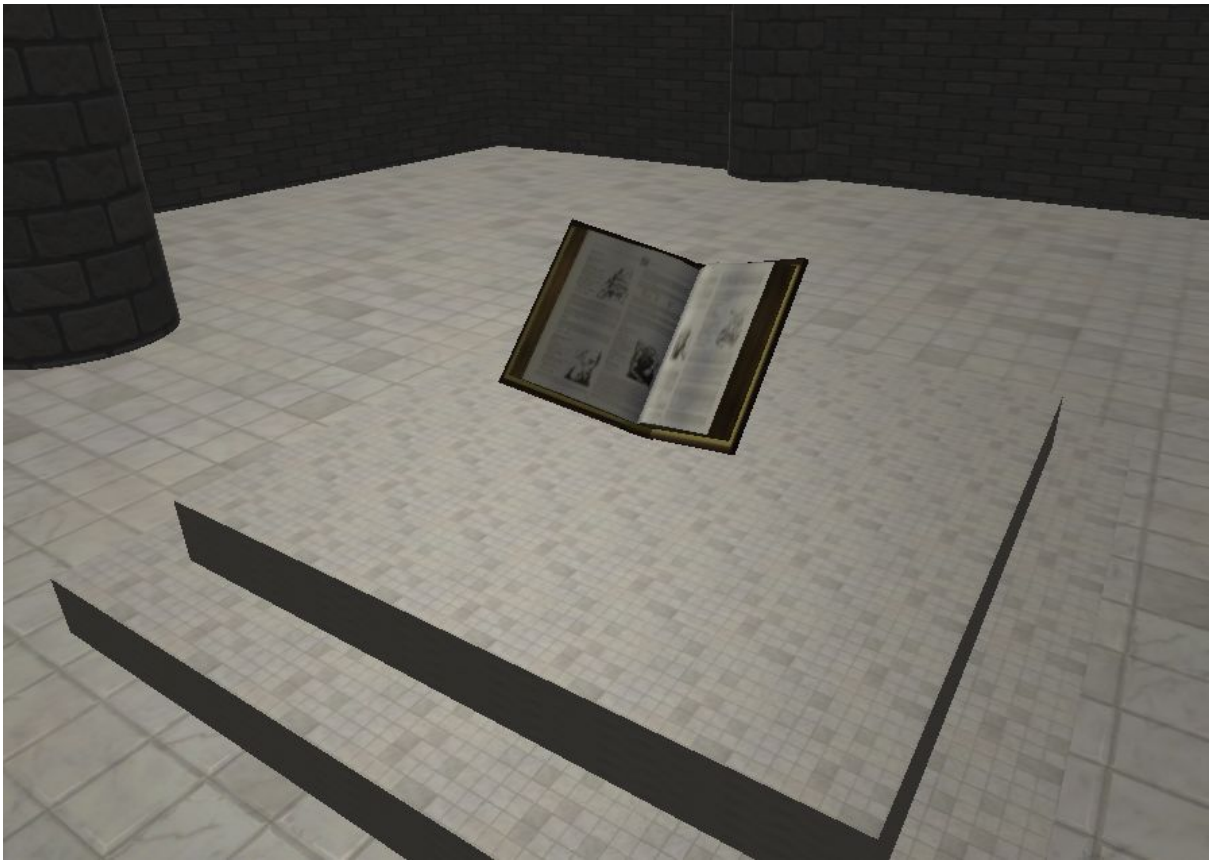




```
//porta da primeira sala

if(altura_porta1 >= -10.0f){
    altura_porta1 = altura_porta1 - 2.0f * deltatime;
}
model_parede = Matrix_Translate(0.0f,altura_porta1,25.0f)
               * Matrix_Rotate_X(1.57f)
               * Matrix_Scale(10.0f,0.5f,15.0f);
glUniformMatrix4fv(model_uniform, 1 , GL_FALSE , glm::value_ptr(model_parede));
glUniform1i(object_id_uniform, PAREDEP);
DrawVirtualObject("parede");
```

Câmeras virtuais:

A câmera livre foi implementada para criar o jogo com perspectiva em primeira pessoa, sem variação de altura pois não queremos que o jogador possa voar pelo cenário, usada para fazer as imagens acima. Um modelo de câmera look-at foi implementado ao concluir o objetivo do jogo como na imagem abaixo.

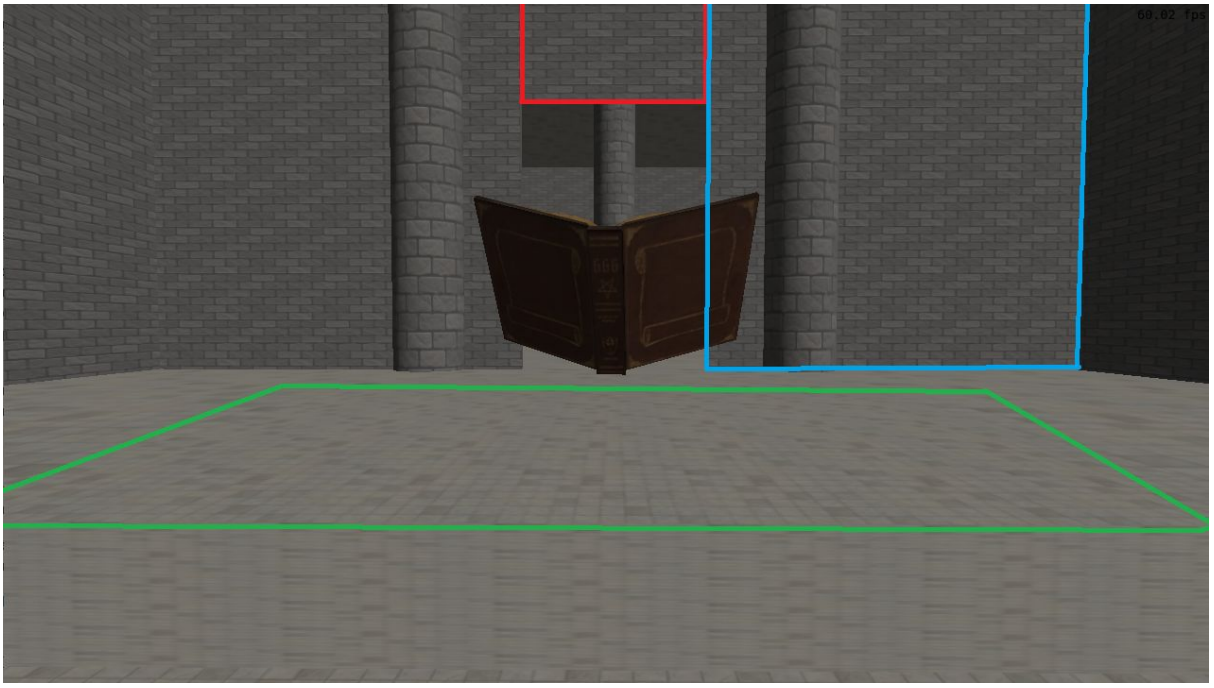


```
//ativa camera look_at ao redor do livro
else{
   if(inicializa_look_camera){
    g_CameraDistance = 10.0f;
    inicializa_look_camera = false;
   }

    camera_position_c  = glm::vec4(-50.0f+x,1.0f+y,100.0f+z,1.0f);
    glm::vec4 camera_lookat_l    = glm::vec4(-50.0f,0.5f,100.0f,1.0f);
    glm::vec4 camera_view_vector = camera_lookat_l - camera_position_c;
    glm::vec4 camera_up_vector   = glm::vec4(0.0f,1.0f,0.0f,0.0f);
```

Mapeamento de texturas em malhas poligonais, múltiplas instâncias de objetos transformados:

Todos os objetos 3D do jogo possuem uma textura apropriada para criar a ambientação desejada, como visto nas imagens anteriores e por exemplo no livro abaixo.

Múltiplas instâncias de um mesmo cubo e um cilindro foram utilizadas para criar o cenário, destacadas em vermelho, verde e azul, temos o mesmo cubo transformado para criar paredes, paredes menores, altares, chão e teto do cenário.



```
//CHÃO
glm::mat4 model_cube = Matrix_Translate(0.0f,-2.1f,0.0f)
                     * Matrix_Scale(50.0f,0.5f,50.0f);
glUniformMatrix4fv(model_uniform, 1 , GL_FALSE , glm::value_ptr(model_cube));
glUniform1i(object_id_uniform, CUBE);
DrawVirtualObject("cube");
```

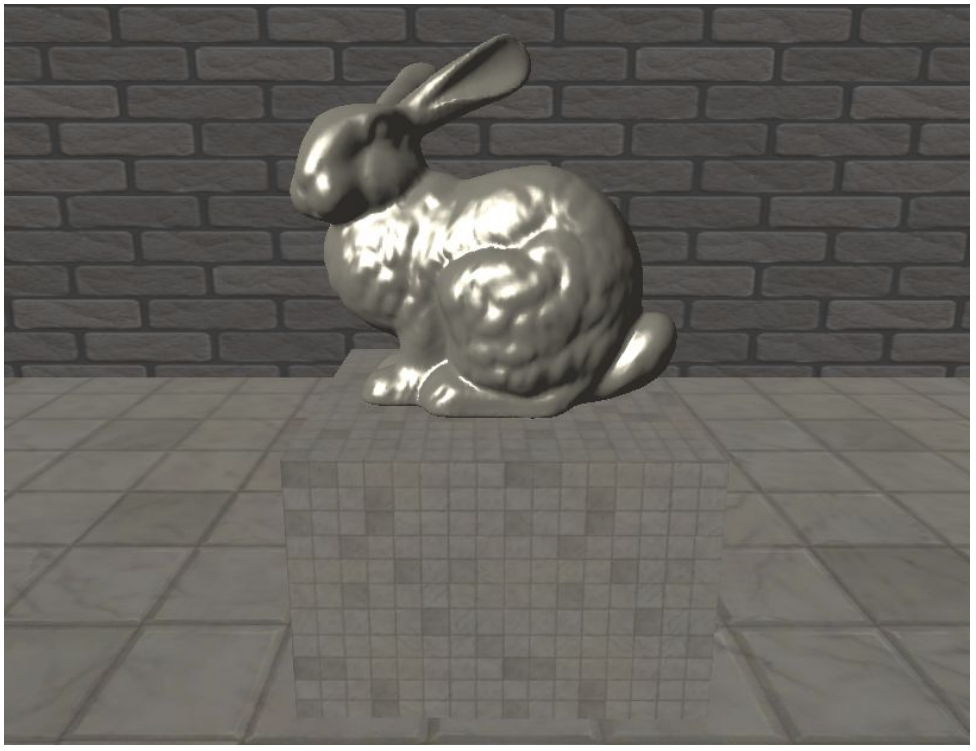
```
//primeiras paredes da porta
model_parede = Matrix_Translate(-15.0f,10.0f,25.0f)
             * Matrix_Rotate_X(1.57f)
             * Matrix_Scale(20.0f,0.5f,25.0f);
glUniformMatrix4fv(model_uniform, 1 , GL_FALSE , glm::value_ptr(model_parede));
glUniform1i(object_id_uniform, PAREDEM);
DrawVirtualObject("cube");
```

```
//altar
model_cube = Matrix_Translate(-50.0f,-1.6f,100.0f)
           * Matrix_Scale(10.0f,1.0f,10.0f);
glUniformMatrix4fv(model_uniform, 1 , GL_FALSE , glm::value_ptr(model_cube));
glUniform1i(object_id_uniform, CUBE);
DrawVirtualObject("cube");
```

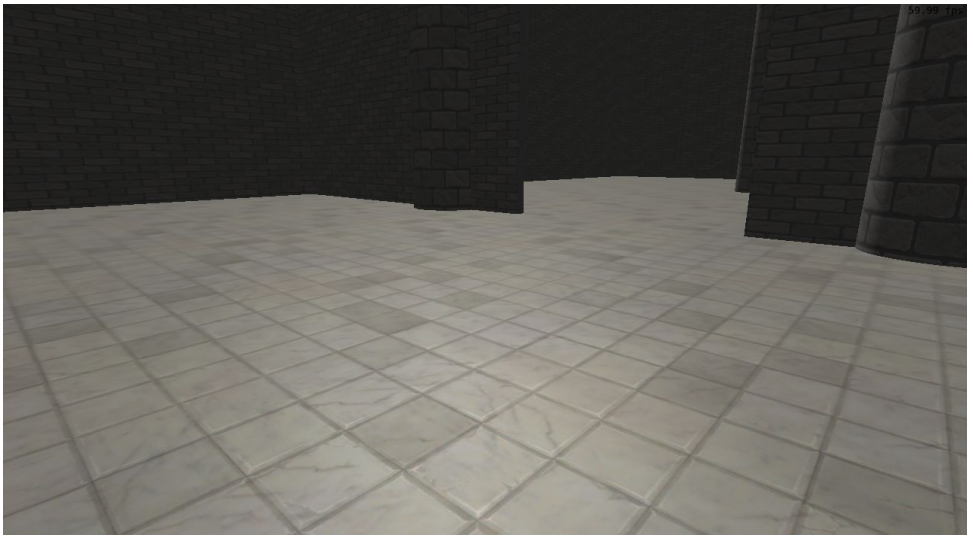
Blinn-Phong shading e Lambert:

Todos os objetos do jogo utilizam pelo menos o termo especular de lambert. Alguns modelos específicos utilizam também o modelo de Blinn-Phong.

Phong shading exagerado em um modelo, com sombreamento usando lambert, apenas para demonstração:



Efeito da iluminação de Phong adicionado apenas ao chão do cenário para dar impressão de lajota com reflexão leve, com coeficientes diferentes dos do coelho de demonstração:



```
color = Kd0 * I * (lambert + 0.01);

//termo de phong para adicionar reflexo nos objetos desejados
vec3 luz = vec3(1.0,1.0,1.0);
vec3 phong = (lambert + 0.1) + I * luz * pow(max(0, angulo_incidencia),q);
```

Gouraud Shading e Lambert:

Usado para iluminar somente os inimigos.

Vertex Shader:

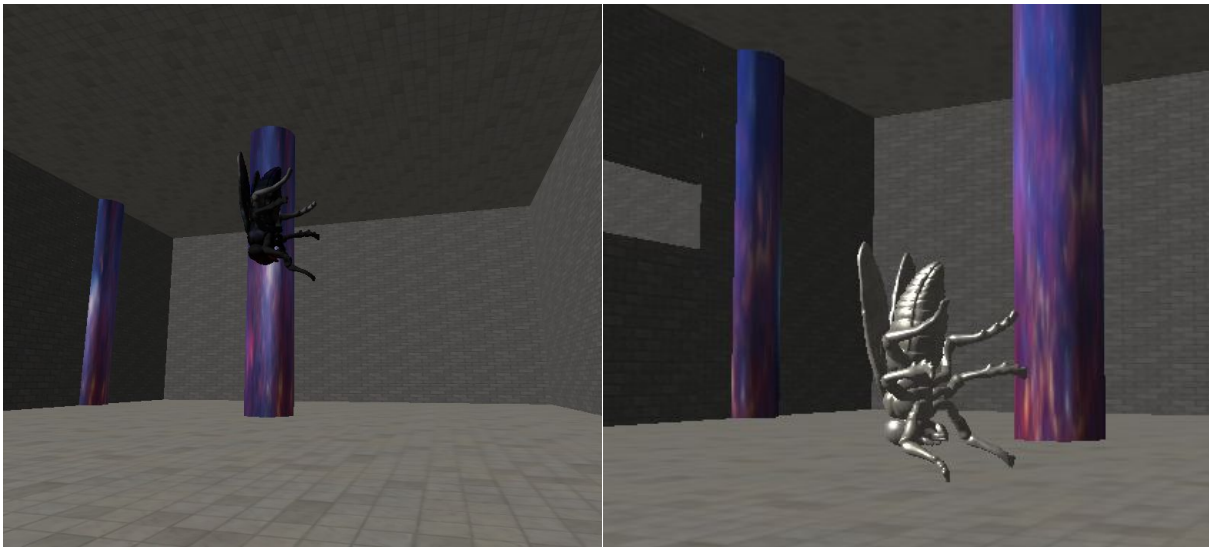
```
vec3 Kd = vec3(0.9, 0.9, 0.9);
vec3 Ka = vec3(0.5,0.5,0.5);
vec3 Ks = vec3(1.0,1.0,1.0);
float q = 1.0;
vec4 origem = vec4(0.0, 0.0, 0.0, 1.0);
vec4 p = position_world;
vec4 camera_position = inverse(view) * origem;
vec4 l = normalize(vec4(1.0,1.0,1.0,0.0));
vec4 n = normalize(normal);
vec4 v = normalize(camera_position - p);
vec4 r = -l+2*n*dot(n,l);


vec3 I = vec3(0.9,0.9,0.9); //fonte de luz
vec3 Ia = vec3(0.05,0.05,0.05);  //luz ambiente
vec3 lambert = Kd*I*max(0, dot(n,l)); //termo difuso de Lambert
vec3 ambient = Ka*Ia; //ambiente
vec3 gourad  = Ks*I* pow(max(0,dot(r,v)),q); //termo de gourad

colorg = lambert + ambient + gourad;
```

Fragment Shader:

```
}
else if(object_id == FLY)
{
    color = texture(TextureImage4, vec2(U,V)).rgb * colorg;
}
```

Intersecções:

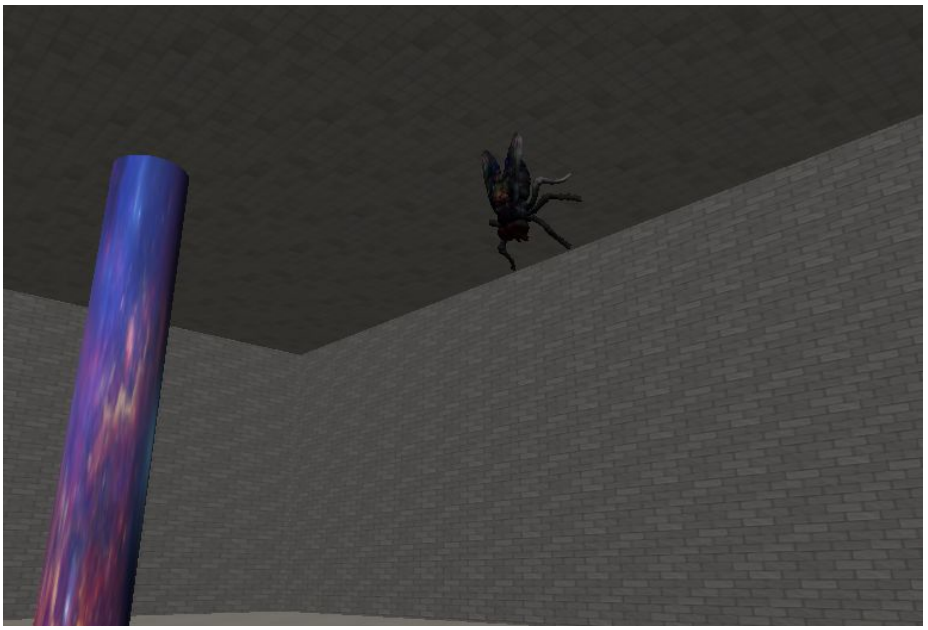
Colisões são calculadas usando os modelos: cubo-cubo para o jogador e o inimigo e coelho de demonstração, cubo-plano para o jogador e a mosca contra o chão e paredes e vetor-cubo para transformar o inimigo em pedra, projetando um vetor da câmera até um objeto.

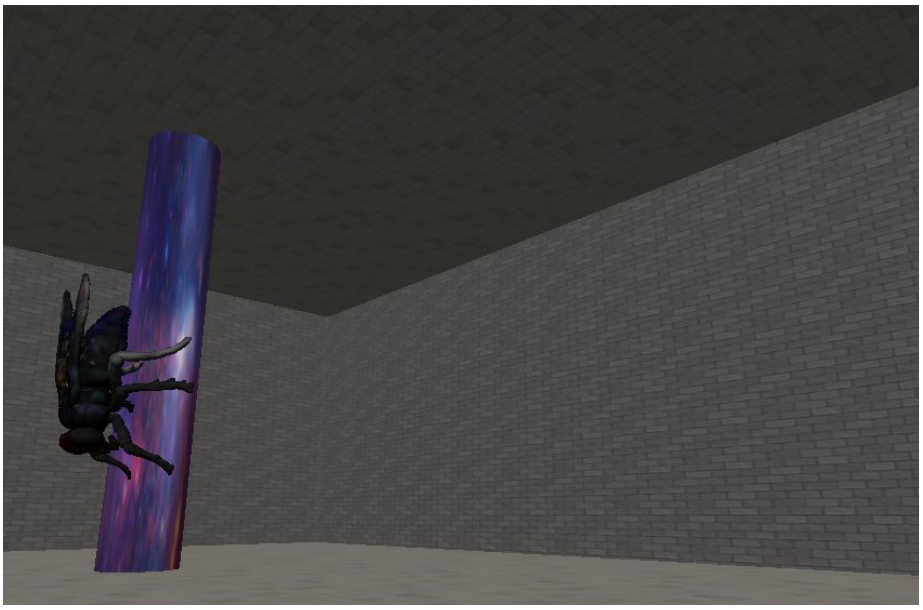


```
bool _colisao(std::string obj1, std::string obj2, int indice)
{
    SceneObject a = g_VirtualScene[obj1];
    SceneObject b = g_VirtualScene[obj2];

    glm::mat4 modelA = a.model[indice];
    glm::mat4 modelB = b.model[indice];

    glm::vec4 a_new_min = aplica_transformacoes_min(a, modelA);
    glm::vec4 a_new_max = aplica_transformacoes_max(a, modelA);
    glm::vec4 b_new_min = aplica_transformacoes_min(b, modelB);
    glm::vec4 b_new_max = aplica_transformacoes_max(b, modelB);



    return(a_new_max.x > b_new_min.x &&
           a_new_min.x < b_new_max.x &&
           a_new_max.y > b_new_min.y &&
           a_new_min.y < b_new_max.y &&
           a_new_max.z > b_new_min.z &&
           a_new_min.z < b_new_max.z);

}
```

Curvas de Bézier cúbica:

Uma curva de Bézier cúbica foi usada para movimentar o inimigo pelo cenário.

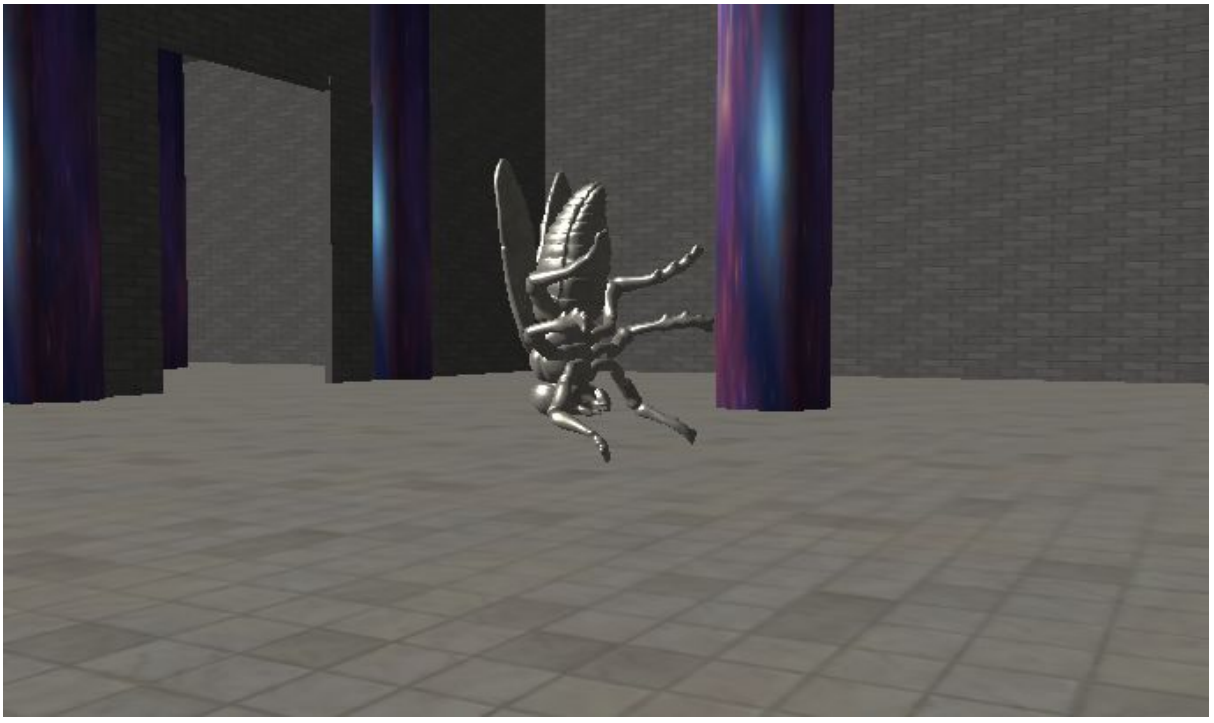


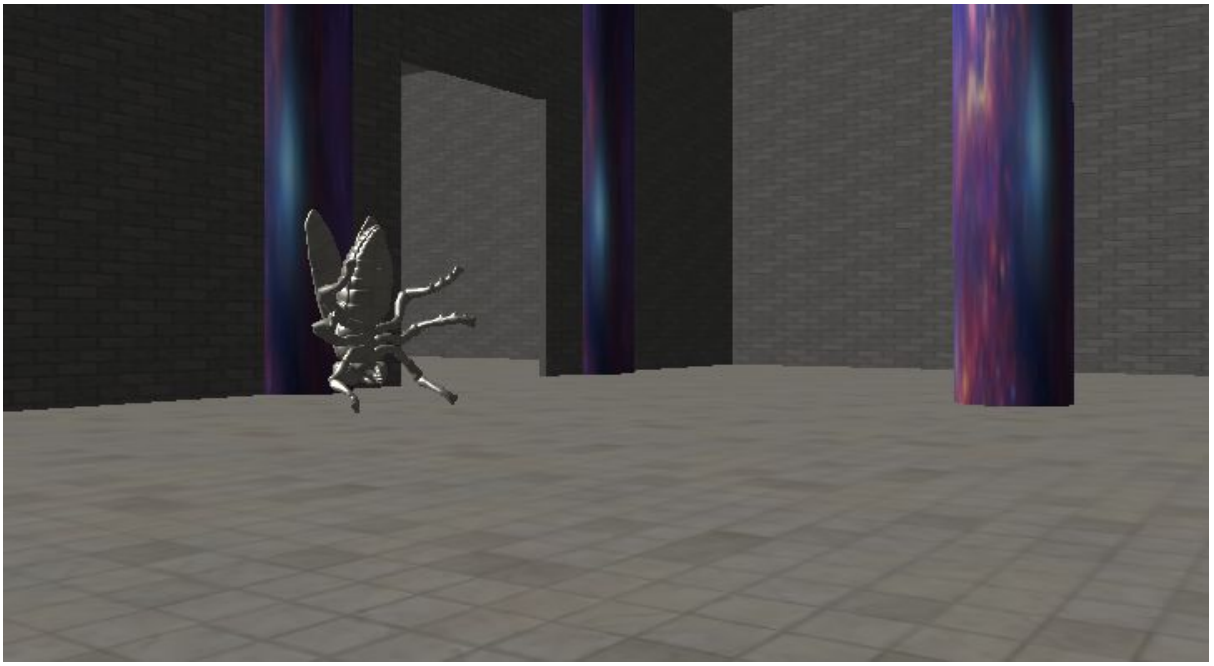


```
void UpdateBezierMovement(float delta_time){
    t_bezier += t_increment*delta_time;
    if(t_bezier < 0 || t_bezier > 1)
    {
        t_increment *= -1;
        t_bezier += t_increment*delta_time;
    }
    pontos_bezier = bezier_p0*(1-t_bezier)*(1-t_bezier)*(1-t_bezier) + 3*t_bezier*(1-t_bezier)*(1-t_bezier)*bezier_p1 +
    (3*t_bezier)*(t_bezier)*(1-t_bezier)*bezier_p2 + (t_bezier)*(t_bezier)*(t_bezier)*bezier_p3;
}
```

Transformações controladas pelo usuário:

São implementadas para poder empurrar inimigos petrificados, segurando-se o botão esquerdo do mouse enquanto olhando para um inimigo petrificado é possível empurrar ele pelo cenário.





```
else if(textura_fly == PHONG && g_LeftMouseButtonPressed && !clique(camera_position_c, camera_view_vector).compare("fly") &&  g_VirtualScene["fly"].clicado)
    if(z>0)
    {
        z_fly = z_fly + 1.0f;
    }
    else
    {
        z_fly =  z_fly - 1.0f;
    }
```

Contribuições individuais:

Mateus Luiz Salvi:

-Freecamera que não pode voar, estilo jogo em primeira pessoa, com efeito da gravidade.
-Mouse restrito à janela, invisível e sem necessidade de manter o botão esquerdo apertado.
-Modelo de iluminação Phong
-Modelo de iluminação Gouraud
-Camera look-at ao chegar no objetivo do jogo
-Animações em função do tempo para as portas e movimentação do jogador.
-Projeção de texturas
-Modelagem do cenário
-Texto de objetivo do jogo animado e de fim de jogo
-Esta documentação
-Vídeo de apresentação
-Curva de bézier para o inimigo

Rafael Oleques Nunes:

- Conversão do AABB em coordenadas do objeto em coordenadas do mundo;
- Colisão cubo-cubo por meio do AABB do objeto da câmera e dos demais objetos da cena, cubo-plano por meio da limitação da área das paredes em relação ao objeto da câmera (não interceptando as áreas das portas) e cubo-vetor por meio de um vetor tendo origem na câmera e direcionado para onde a câmera está virada, verificando a intersecção com outros objetos (escolhidos previamente como interativos);
- Câmera envolta a um cubo deformado para realizar as colisões;
- Reconhecimento de um objeto que está sendo clicado, dentre os que previamente foram escolhidos como interativos;
- Modificação da textura dos inimigos (aqueles que foram escolhidos como interativos) ao realizar uma colisão cubo-vetor, sendo ela feita por meio do clique do mouse em cima deles;
- Movimentar os objetos inimigos (aqueles que foram escolhidos como interativos) no eixo z global por meio de uma colisão cubo-vetor, no mesmo sentido em que a câmera está virada, depois que eles já tiveram uma colisão cubo-vetor anteriormente;
- Animação em função do tempo dos cilindros *de energia* que estão no cenário.