Lecture 5

Nesta parte da disciplina apresentamos um algoritmo de preenchimento de triângulo usando interpolação de cores.

Coordenadas Baricêntricas

Sejam $C_0 = (x_0, y_0)$, $C_1 = (x_1, y_1)$ e $C_2 = (x_2, y_2)$ pontos não colineares no espaço 2D. Podemos realizar o preenchimento do polígono (triângulo) definido entre esses 3 pontos usando a equação: $C = \alpha C_0 + \beta C_1 + \gamma C_2$, onde $\alpha + \beta + \gamma = 1$.

Figure 8. Preenchimento de triângulos usando coordenadas baricêntricas

Código

A implementação do algoritmo de coordenadas baricêntricas está disponível no arquivo barycentric.c.

```c
/**
 * \file barycentric.c
 *
 * \brief Implementação da renderizacao de modelo 3D
 *
 * \author
 * Petrucio Ricardo Tavares de Medeiros \n
 * Universidade Federal Rural do Semi-Arido \n
 * Departamento de Engenharias e Tecnologia \n
 * petrucio at ufersa (dot) edu (dot) br
 *
 * \version 1.0
 * \date Jul 2025
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define WIDTH 800
#define HEIGHT 800
#define MAX_VERTICES 50000
#define MAX_FACES 50000
#define MAX_FACE_VERTS 32

typedef struct {
  float x, y, z;
} Vertex;

typedef struct {
  int verts[MAX_FACE_VERTS];
  int n;
} Face;
```

```
unsigned char image[WIDTH][HEIGHT][3];

void set_pixel(int x, int y, unsigned char r, unsigned char g, unsigned char b) {
  if (x >= 0 && x < WIDTH && y >= 0 && y < HEIGHT) {
    image[y][x][0] = r;
    image[y][x][1] = g;
    image[y][x][2] = b;
  }
}

void draw_line(int x0, int y0, int x1, int y1) {
  for (float t = 0.0; t < 1.0; t = t + 0.0001)
    set_pixel((int)x0+(x1-x0)*t, (int)y0+(y1-y0)*t, 0, 0, 0);
}

void clr(){
  for(int i = 0; i < WIDTH; i++)
    for(int j = 0; j < HEIGHT; j++)
      for(int c = 0; c < 3; c++)
        image[i][j][c] = 255;
}

void save(){
  printf("P3\n %d \t %d\n 255\n", WIDTH, HEIGHT);
  for(int i = 0; i < WIDTH; i++){
    for(int j = 0; j < HEIGHT; j++){
      for(int c = 0; c < 3; c++){
        printf("%d \t", image[i][j][c]);
      }
      printf("\n");
    }
  }
}

int load_obj(const char *filename, Vertex *vertices, int *vcount, Face *faces,
             int *fcount) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("Erro ao abrir o arquivo");
        return 0;
    }

    char line[512];
    *vcount = 0;
    *fcount = 0;

    while (fgets(line, sizeof(line), file)) {
        if (strncmp(line, "v ", 2) == 0) {
            if (sscanf(line + 2, "%f %f %f", &vertices[*vcount].x,
                       &vertices[*vcount].y, &vertices[*vcount].z) == 3) {
                (*vcount)++;
            }
        } else if (strncmp(line, "f ", 2) == 0) {
            Face face = {.n = 0};
            char *token = strtok(line + 2, " \n");
            while (token && face.n < MAX_FACE_VERTS) {
                int index;
                if (sscanf(token, "%d", &index) == 1) {
                    face.verts[face.n++] = index;
                }
                token = strtok(NULL, " \n");
            }
            faces[(*fcount)++] = face;
        }
    }

    fclose(file);
    return 1;
```

```
}

void resizing( Vertex v0, Vertex v1 ){
  int x0 = (int)((v0.x + 1.0f) * WIDTH / 2.0f);
  int y0 = (int)((1.0f - v0.y) * HEIGHT / 2.0f);
  int x1 = (int)((v1.x + 1.0f) * WIDTH / 2.0f);
  int y1 = (int)((1.0f - v1.y) * HEIGHT / 2.0f);

  draw_line(x0, y0, x1, y1);
}


void render_faces(Vertex *vertices, Face *faces, int vcount, int fcount) {
  for (int i = 0; i < fcount; i++) {
    Face face = faces[i];
    for (int j = 0; j < face.n; j++) {
      Vertex v0 = vertices[face.verts[j] - 1];
      Vertex v1 = vertices[face.verts[(j + 1) % face.n] - 1];
      resizing(v0, v1);
    }
  }
}

void rotate_z(Vertex *v, float angle_rad) {
    float x = v->x;
    float y = v->y;
    v->x = x * cosf(angle_rad) - y * sinf(angle_rad);
    v->y = x * sinf(angle_rad) + y * cosf(angle_rad);
}

void project_3dto2d(Vertex *v) {
    v->x = (v->x + 1.0f) * (WIDTH  / 2.0f);
    v->y = (1.0f + v->y) * (HEIGHT / 2.0f);
}

void barycentric_coordinate( Vertex a, Vertex b, Vertex c, float red, float green, float blue ){
  // calculando o bounding box
  int x_min = floorf(fminf(fminf(a.x, b.x), c.x));
  int x_max = ceilf(fmaxf(fmaxf(a.x, b.x), c.x));
  int y_min = floorf(fminf(fminf(a.y, b.y), c.y));
  int y_max = ceilf(fmaxf(fmaxf(a.y, b.y), c.y));

  // Encontrando a área do triangulo abc
  float area_abc = 0.5 * fabsf(a.x*(b.y - c.y) + b.x*(c.y - a.y) + c.x*(a.y - b.y));

  for (int y = y_min; y < y_max; y++){
    for (int x = x_min; x < x_max; x++){
      Vertex p = {x, y, 0};
      // Encontrando a área dos triangulos
      float area_pbc = 0.5 * (p.x*(b.y - c.y) + b.x*(c.y - p.y) + c.x*(p.y - b.y));
      float area_apc = 0.5 * (a.x*(p.y - c.y) + p.x*(c.y - a.y) + c.x*(a.y - p.y));
      float area_abp = 0.5 * (a.x*(b.y - p.y) + b.x*(p.y - a.y) + p.x*(a.y - b.y));
      float alfa = area_pbc / area_abc;
      float beta = area_apc / area_abc;
      float gamma = area_abp / area_abc;
      if ( alfa >= 0.0 && beta >= 0.0 && gamma >= 0.0 ){
        set_pixel( x, y, red, green, blue );
      }
    }
  }
}

void render_faces_filled( Vertex *vertices, Face *faces, int vcount, int fcount){
  for (int i = 0; i < fcount; i++){
    Face face = faces[i];

    Vertex v0 = vertices[face.verts[0] - 1];
    Vertex v1 = vertices[face.verts[1] - 1];
    Vertex v2 = vertices[face.verts[2] - 1];
```

```
    // Rotacione os vertices (180 graus)
    rotate_z(&v0, M_PI);
    rotate_z(&v1, M_PI);
    rotate_z(&v2, M_PI);

    // Projecao 3D -> 2D
    project_3dto2d(&v0);
    project_3dto2d(&v1);
    project_3dto2d(&v2);

    barycentric_coordinate( v0, v1, v2, rand()%255, rand()%255, rand()%255 );
  }
}

int main(){

  Vertex vertices[MAX_VERTICES];
  Face faces[MAX_FACES];
  int vcount, fcount;

  clr();

  // Lê o arquivo OBJ enviado
  if (!load_obj("models/wolf.obj", vertices, &vcount, faces, &fcount)) {
    return 1;
  }

  // Renderiza as faces no framebuffer
  render_faces_filled( vertices, faces, vcount, fcount );

  save();

  return 0;
}
```

Descrição do programa

Para evitar redundância, descreveremos apenas o conteúdo que adicionamos ao arquivo. Sendo assim, discutiremos sobre a função **barycentric_coordinate**. Esta função recebe 3 vértices e calcula o *bounding box*, ou seja, o volume que envolve o triângulo que desejamos preencher. Em seguida, encontramos as áreas dos triângulos e depois percorremos todos os pixels definido no *bounding box* com intuito de investigar se os alfa, beta e gamma são maiores ou iguais a zero para pintar o pixel.

```
void barycentric_coordinate( Vertex a, Vertex b, Vertex c ){
  // calculando o bounding box
  int x_min = floorf(fminf(fminf(a.x, b.x), c.x));
  int x_max = ceilf(fmaxf(fmaxf(a.x, b.x), c.x));
  int y_min = floorf(fminf(fminf(a.y, b.y), c.y));
  int y_max = ceilf(fmaxf(fmaxf(a.y, b.y), c.y));

  // Encontrando a área do triangulo abc
  float area_abc = 0.5 * fabsf(a.x*(b.y - c.y) + b.x*(c.y - a.y) + c.x*(a.y - b.y));
  float red = rand()%255;
  float green = rand()%255;
  float blue = rand()%255;
  for (int y = y_min; y < y_max; y++){
    for (int x = x_min; x < x_max; x++){
      Vertex p = {x, y, 0};
      // Encontrando a área dos triangulos
      float area_pbc = 0.5 * (p.x*(b.y - c.y) + b.x*(c.y - p.y) + c.x*(p.y - b.y));
      float area_apc = 0.5 * (a.x*(p.y - c.y) + p.x*(c.y - a.y) + c.x*(a.y - p.y));
      float area_abp = 0.5 * (a.x*(b.y - p.y) + b.x*(p.y - a.y) + p.x*(a.y - b.y));
      float alfa = area_pbc / area_abc;
      float beta = area_apc / area_abc;
      float gamma = area_abp / area_abc;
      if ( alfa >= 0.0 && beta >= 0.0 && gamma >= 0.0 ){
```

```
            set_pixel( x, y, red, green, blue );
          }
        }
      }
    }
```

Uma outra função que devemos discutir é a função **render_faces_filled**. Nesta função, encontramos os 3 vértices para uma face de triângulo, transformamos esses vértices e preechemos esses triângulos usando a função **barycentric_coordinate** previamente discutida.

```
void render_faces_filled( Vertex *vertices, Face *faces, int vcount, int fcount){
  for (int i = 0; i < fcount; i++){
    Face face = faces[i];

    Vertex v0 = vertices[face.verts[0] - 1];
    Vertex v1 = vertices[face.verts[1] - 1];
    Vertex v2 = vertices[face.verts[2] - 1];

    // Rotacione os vertices (180 graus)
    rotate_z(&v0, M_PI);
    rotate_z(&v1, M_PI);
    rotate_z(&v2, M_PI);

    // Projecao 3D -> 2D
    project_3dto2d(&v0);
    project_3dto2d(&v1);
    project_3dto2d(&v2);

    barycentric_coordinate( v0, v1, v2 );
  }
}
```

Lecture 6

Vamos aprender como a iluminação é calculada em computação gráfica para simular a interação da luz com superfícies 3D.

Iluminação

A iluminação em computação gráfica simula como a luz interage com as superfícies dos objetos para gerar realismo visual. Os três principais componentes são: **ambiente**, que representa a luz difusa constante do ambiente; **difusa**, que depende do ângulo entre a luz e a superfície; e **especular**, que simula o brilho refletido em direção ao observador. Esses elementos compõem o modelo clássico de Phong, amplamente usado por seu equilíbrio entre realismo e eficiência. A intensidade final da cor é calculada somando essas contribuições, considerando também os materiais e a direção da câmera.

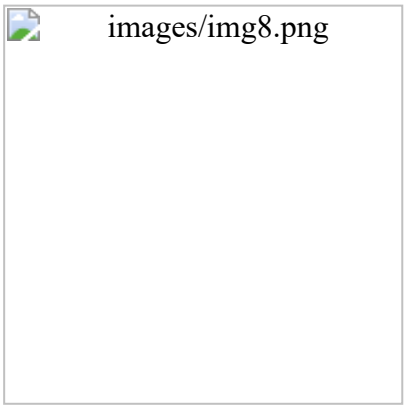

Figure 9. Resultado da iluminação de Phong

Reflexão ambiente

$I_{amb} = k_a I_{ambiente}$

- k_a: coeficiente de reflexão ambiente do material (0 a 1);
- $I_{ambiente}$: intensidade da luz ambiente

Reflexão difusa (Lambert)

$I_{diff} = k_d I_L max(0, \vec{N} . \vec{L})$

- k_d: coeficiente de reflexão difuso;
- I_{L}: intensidade da luz;
- $\vec{N}$: vetor normal da superfície;
- $\vec{L}$: vetor da direção da luz
- produto escalar: mede o "alinhamento" da luz com a superfície

Reflexão especular (Phong)

$I_{spec} = k_s I_L max(0, \vec{R} . \vec{V})^n$, onde $\vec{R} = 2(\vec{N}.\vec{L})\vec{N} - \vec{L}$

- k_s: coeficiente especular;
- I_{L}: intensidade da luz;
- $\vec{R}$: vetor de reflexão da luz;
- $\vec{V}$: vetor em direção à câmera
- n: brilho (quanto maior, mais estreito o brilho)

Código

A implementação da iluminação está disponível no arquivo light.c.

```c
/**
 * \file light.c
 *
 * \brief Implementação da iluminação com reflexão ambiente, difusa (lambert)
 * e especular (phong)
 *
 * \author
 * Petrucio Ricardo Tavares de Medeiros \n
 * Universidade Federal Rural do Semi-Arido \n
 * Departamento de Engenharias e Tecnologia \n
 * petrucio at ufersa (dot) edu (dot) br
 *
 * \version 1.0
 * \date Jul 2025
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define WIDTH 800
#define HEIGHT 800
```

```c
#define MAX_VERTICES 50000
#define MAX_FACES 50000
#define MAX_FACE_VERTS 32

typedef struct {
  float x, y, z;
} Vertex;

typedef struct {
  int verts[MAX_FACE_VERTS];
  int n;
} Face;

unsigned char image[WIDTH][HEIGHT][3];

void set_pixel(int x, int y, unsigned char r, unsigned char g, unsigned char b) {
  if (x >= 0 && x < WIDTH && y >= 0 && y < HEIGHT) {
    image[y][x][0] = r;
    image[y][x][1] = g;
    image[y][x][2] = b;
  }
}

void draw_line(int x0, int y0, int x1, int y1) {
  for (float t = 0.0; t < 1.0; t = t + 0.0001)
    set_pixel((int)x0+(x1-x0)*t, (int)y0+(y1-y0)*t, 0, 0, 0);
}

void clr(){
  for(int i = 0; i < WIDTH; i++)
    for(int j = 0; j < HEIGHT; j++)
      for(int c = 0; c < 3; c++)
        image[i][j][c] = 255;
}

void save(){
  printf("P3\n %d \t %d\n 255\n", WIDTH, HEIGHT);
  for(int i = 0; i < WIDTH; i++){
    for(int j = 0; j < HEIGHT; j++){
      for(int c = 0; c < 3; c++){
        printf("%d \t", image[i][j][c]);
      }
      printf("\n");
    }
  }
}

int load_obj(const char *filename, Vertex *vertices, int *vcount, Face *faces,
             int *fcount) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("Erro ao abrir o arquivo");
        return 0;
    }

    char line[512];
    *vcount = 0;
    *fcount = 0;

    while (fgets(line, sizeof(line), file)) {
        if (strncmp(line, "v ", 2) == 0) {
            if (sscanf(line + 2, "%f %f %f", &vertices[*vcount].x,
                       &vertices[*vcount].y, &vertices[*vcount].z) == 3) {
                (*vcount)++;
            }
        } else if (strncmp(line, "f ", 2) == 0) {
            Face face = {.n = 0};
            char *token = strtok(line + 2, " \n");
            while (token && face.n < MAX_FACE_VERTS) {
```

```
                int index;
                if (sscanf(token, "%d", &index) == 1) {
                    face.verts[face.n++] = index;
                }
                token = strtok(NULL, " \n");
            }
            faces[(*fcount)++] = face;
        }
    }

    fclose(file);
    return 1;
}

void rotate_z(Vertex *v, float angle_rad) {
    float x = v->x;
    float y = v->y;
    v->x = x * cosf(angle_rad) - y * sinf(angle_rad);
    v->y = x * sinf(angle_rad) + y * cosf(angle_rad);
}

void project_3dto2d(Vertex *v) {
    v->x = (v->x + 1.0f) * (WIDTH  / 2.0f);
    v->y = (1.0f + v->y) * (HEIGHT / 2.0f);
}

void barycentric_coordinate( Vertex a, Vertex b, Vertex c, float red, float green, float blue ){
  // calculando o bounding box
  int x_min = floorf(fminf(fminf(a.x, b.x), c.x));
  int x_max = ceilf(fmaxf(fmaxf(a.x, b.x), c.x));
  int y_min = floorf(fminf(fminf(a.y, b.y), c.y));
  int y_max = ceilf(fmaxf(fmaxf(a.y, b.y), c.y));

  // Encontrando a área do triangulo abc
  float area_abc = 0.5 * fabsf(a.x*(b.y - c.y) + b.x*(c.y - a.y) + c.x*(a.y - b.y));

  for (int y = y_min; y <= y_max; y++){
    for (int x = x_min; x <= x_max; x++){
      Vertex p = {x, y, 0};
      // Encontrando a área dos triangulos
      float area_pbc = 0.5 * (p.x*(b.y - c.y) + b.x*(c.y - p.y) + c.x*(p.y - b.y));
      float area_apc = 0.5 * (a.x*(p.y - c.y) + p.x*(c.y - a.y) + c.x*(a.y - p.y));
      float area_abp = 0.5 * (a.x*(b.y - p.y) + b.x*(p.y - a.y) + p.x*(a.y - b.y));
      float alfa = area_pbc / area_abc;
      float beta = area_apc / area_abc;
      float gamma = area_abp / area_abc;
      if ( alfa >= 0.0 && beta >= 0.0 && gamma >= 0.0 ){
        set_pixel( x, y, red, green, blue );
      }
    }
  }
}

Vertex sub( Vertex a, Vertex b ){
  return (Vertex) {a.x - b.x, a.y - b.y, a.z - b.z};
}

float dot( Vertex a, Vertex b ){
  return a.x * b.x + a.y * b.y + a.z * b.z;
}

Vertex cross( Vertex a, Vertex b ){
  return (Vertex) {a.y*b.z - a.z*b.y, a.z*b.x - a.x*b.z, a.x*b.y - a.y*b.x};
}

Vertex normalize( Vertex v ){
  float len = sqrtf(v.x*v.x + v.y*v.y + v.z*v.z);
  if ( len == 0 ) return (Vertex){0, 0, 0};
  return (Vertex){v.x/len, v.y/len, v.z/len};
```

```
}

Vertex scalar( float x, Vertex v ){
  return (Vertex){ x*v.x, x*v.y, x*v.z };
}

void render_faces_filled( Vertex *vertices, Face *faces, int vcount, int fcount ){

  Vertex light = {0, 0, -1}; //{0.25, 0.0, -0.75};
  Vertex view_dir = {0, 0, 1}; // Camera olhando para -z

  for (int i = 0; i < fcount; i++){
    Face face = faces[i];

    Vertex v0 = vertices[face.verts[0] - 1];
    Vertex v1 = vertices[face.verts[1] - 1];
    Vertex v2 = vertices[face.verts[2] - 1];

    // Vetor normal
    Vertex v1_v0 = sub( v1, v0 );
    Vertex v2_v0 = sub( v2, v0 );
    Vertex normal = normalize( cross( v2_v0, v1_v0 ) );

    // Coeficientes de phong
    float ka = 0.2; // ambiente
    float kd = 0.6; // difusa
    float ks = 0.4; // especular
    int brilho = 32;

    // Normalizar luz e direcao da camera
    Vertex L = normalize(light);
    Vertex V = normalize(view_dir);

    // Iluminacao difusa
    float diff = fmaxf(0, dot( normal, L ));

    // Iluminacao especular
    Vertex R = sub( scalar(2.0 * dot(normal,L), normal), L); // R = 2(N.L)N - L
    float spec = powf(fmaxf(0, dot(R, V)), brilho);

    // Intensidade final
    float intensity = ka + kd * diff + ks * spec;
    if ( intensity > 1.0 ) intensity = 1.0;

    // Rotacione os vertices (180 graus)
    rotate_z(&v0, M_PI);
    rotate_z(&v1, M_PI);
    rotate_z(&v2, M_PI);

    // Projecao 3D -> 2D
    project_3dto2d(&v0);
    project_3dto2d(&v1);
    project_3dto2d(&v2);

    barycentric_coordinate( v0, v1, v2, intensity*255, intensity*255, intensity*255 );
  }
}

int main(){

  Vertex vertices[MAX_VERTICES];
  Face faces[MAX_FACES];
  int vcount, fcount;

  clr();

  // Lê o arquivo OBJ enviado
  if (!load_obj("models/wolf.obj", vertices, &vcount, faces, &fcount)) {
    return 1;
```

```
  }

  // Renderiza as faces no framebuffer
  render_faces_filled( vertices, faces, vcount, fcount );

  save();

  return 0;
}
```

Descrição do programa

Neste código, implementamos funções auxiliares para o cálculo da iluminação, como subtração de vetores (**sub**), produto escalar (**dot**), produto vetorial (**cross**), normalização (**normalize**) e multiplicação por escalar (**scalar**). Essas funções são utilizadas na função **render_faces_filled**, responsável por aplicar o modelo de iluminação sobre os triângulos da malha. A partir dos vertices $v0$, $v1$ e $v2$ de cada triângulo, calculamos a normal da face e definimos os coeficientes de iluminação. Em seguida, normalizamos os vetores da luz e do observador, computamos as componentes difusa e especular, e combinamos os resultados para determinar a intensidade final da cor em cada triângulo renderizado.

```
void render_faces_filled( Vertex *vertices, Face *faces, int vcount, int fcount ){

  Vertex light = {0, 0, -1}; //{0.25, 0.0, -0.75};
  Vertex view_dir = {0, 0, 1}; // Camera olhando para -z

  for (int i = 0; i < fcount; i++){
    Face face = faces[i];

    Vertex v0 = vertices[face.verts[0] - 1];
    Vertex v1 = vertices[face.verts[1] - 1];
    Vertex v2 = vertices[face.verts[2] - 1];

    // Vetor normal
    Vertex v1_v0 = sub( v1, v0 );
    Vertex v2_v0 = sub( v2, v0 );
    Vertex normal = normalize( cross( v2_v0, v1_v0 ) );

    // Coeficientes de phong
    float ka = 0.2; // ambiente
    float kd = 0.6; // difusa
    float ks = 0.4; // especular
    int brilho = 32;

    // Normalizar luz e direcao da camera
    Vertex L = normalize(light);
    Vertex V = normalize(view_dir);

    // Iluminacao difusa
    float diff = fmaxf(0, dot( normal, L ));

    // Iluminacao especular
    Vertex R = sub( scalar(2.0 * dot(normal,L), normal), L); // R = 2(N.L)N - L
    float spec = powf(fmaxf(0, dot(R, V)), brilho);

    // Intensidade final
    float intensity = ka + kd * diff + ks * spec;
    if ( intensity > 1.0 ) intensity = 1.0;

    // Rotacione os vertices (180 graus)
    rotate_z(&v0, M_PI);
    rotate_z(&v1, M_PI);
    rotate_z(&v2, M_PI);

    // Projecao 3D -> 2D
    project_3dto2d(&v0);
    project_3dto2d(&v1);
    project_3dto2d(&v2);
```

```
    barycentric_coordinate( v0, v1, v2, intensity*255, intensity*255, intensity*255 );
  }
}
```

Lecture 7

Vamos compreender como a técnica ray tracing funciona.

Ray Tracing

Técnica de síntese de imagens fotorealísticas que traça raios da câmera até a cena. Se um raio intercepta algum objeto da cena, cálculos relacionados a iluminação são realizados. Caso contrário, a cor do background é retornado.

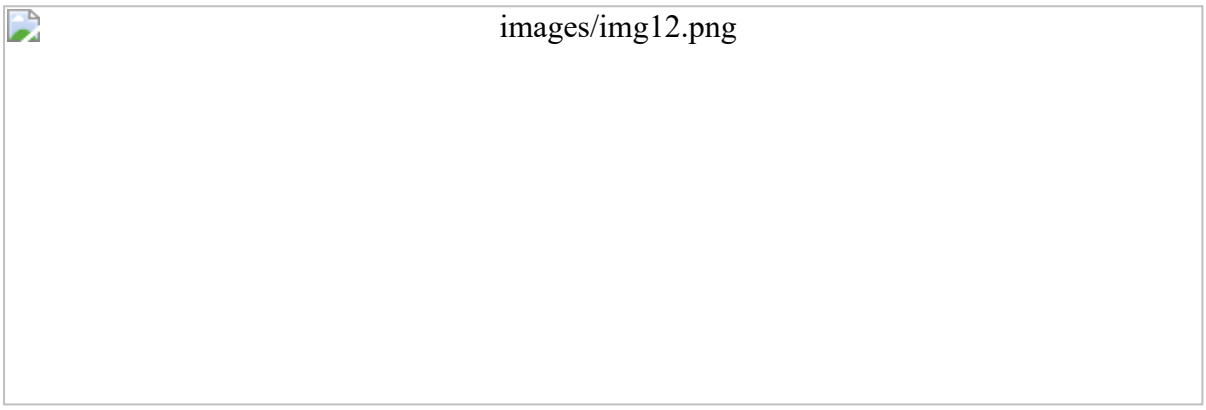

Figure 10. Imagens sintetizadas a partir do algoritmo raytracing.

Códigos

A implementação das imagens estão disponiveis nos arquivos raytracing.c, raytracing2.c e raytracing3.c, respectivamente.

Renderização de uma esfera com background azul

```
/**
 * \file raytracing.c
 *
 * \brief Implementação do traçador de raios para criação de imagens
 * com renderização de uma esfera.
 *
 * \author
 * Petrucio Ricardo Tavares de Medeiros \n
 * Universidade Federal Rural do Semi-Arido \n
 * Departamento de Engenharias e Tecnologia \n
 * petrucio at ufersa (dot) edu (dot) br
 *
 * \version 1.0
 * \date Jul 2025
 */

#include <stdio.h>
#include <math.h>

#define WIDTH 800
#define HEIGHT 600

unsigned char img[WIDTH][HEIGHT][3];

typedef struct{
  float x, y, z;
```

```
} Vertex;

Vertex add( Vertex a, Vertex b ){ return (Vertex){a.x + b.x, a.y + b.y, a.z + b.z}; }
Vertex sub( Vertex a, Vertex b ){ return (Vertex){a.x - b.x, a.y - b.y, a.z - b.z}; }
Vertex scale( Vertex v, float s ){ return (Vertex){s*v.x, s*v.y, s*v.z}; }
float dot(Vertex a, Vertex b){ return a.x*b.x + a.y*b.y + a.z*b.z; }
float length(Vertex v){ return sqrtf(dot(v, v)); }
Vertex normalize( Vertex v ){ return scale(v, 1.0f / length( v )); }

void save(){
  printf("P3\n %d \t %d\n 255\n", WIDTH, HEIGHT);
  for (int y = 0; y < HEIGHT; y++){
    for (int x = 0; x < WIDTH; x++){
      for (int c = 0; c < 3; c++){
        printf("%d \t", img[x][y][c]);
      }
      printf("\n");
    }
  }
}

// Retorna t (distância) ou -1 se não há interseção
float intersecao_esfera( Vertex O, Vertex D, Vertex C, float r ){
  Vertex L = sub( O, C );
  float a = dot( D, D );
  float b = 2.0 * dot(L, D);
  float c = dot(L, L) - r * r;
  float delta = b * b - 4 * a * c;
  if ( delta < 0 ) return -1.0;
  float sqrt_delta = sqrtf( delta );
  float t0 = (-b - sqrt_delta) / (2 * a);
  float t1 = (-b + sqrt_delta) / (2 * a);
  if ( t0 > 0.001 ) return t0;
  if ( t1 > 0.001 ) return t1;
  return -1.0;
}

void render(){
  Vertex camera = (Vertex){0, 0, 0};
  Vertex C = (Vertex){0, 0, -5};
  float r = 1.0;
  Vertex light_position = normalize( (Vertex){1, 1, 1} );

  for (int y = 0; y < HEIGHT; y++){
    for (int x = 0; x < WIDTH; x++){
      float fx = (2.0 * x / WIDTH - 1.0) * (float)WIDTH / HEIGHT;
      float fy = 1.0 - 2.0 * y / (float) HEIGHT;
      Vertex D = normalize( (Vertex){fx, fy, -1} );

      float t = intersecao_esfera(camera, D, C, r);
      if ( t > 0.0 ){
        Vertex hitPoint = add( camera, scale( D, t ));
        Vertex normal = normalize( sub(hitPoint, C) );
        float diff = fmaxf(0.0, dot(normal, light_position));
        img[x][y][0] = (unsigned char)(diff * 255);
        img[x][y][1] = 0;
        img[x][y][2] = 0;
      }
      else{
        img[x][y][0] = 135;
        img[x][y][1] = 206;
        img[x][y][2] = 250;
      }
    }
  }
}

int main(){
  render();
```

```
  save();
  return 0;
}
```

Renderização de uma esfera com plano xadrez

```
/**
 * \file raytracing2.c
 *
 * \brief Implementação do traçador de raios para criação de imagens
 * com renderização de uma esfera e um plano.
 *
 * \author
 * Petrucio Ricardo Tavares de Medeiros \n
 * Universidade Federal Rural do Semi-Arido \n
 * Departamento de Engenharias e Tecnologia \n
 * petrucio at ufersa (dot) edu (dot) br
 *
 * \version 1.0
 * \date Jul 2025
 */

#include <stdio.h>
#include <math.h>

#define WIDTH 800
#define HEIGHT 600

unsigned char img[WIDTH][HEIGHT][3];

typedef struct{
  float x, y, z;
} Vertex;

Vertex add( Vertex a, Vertex b ){ return (Vertex){a.x + b.x, a.y + b.y, a.z + b.z}; }
Vertex sub( Vertex a, Vertex b ){ return (Vertex){a.x - b.x, a.y - b.y, a.z - b.z}; }
Vertex scale( Vertex v, float s ){ return (Vertex){s*v.x, s*v.y, s*v.z}; }
float dot(Vertex a, Vertex b){ return a.x*b.x + a.y*b.y + a.z*b.z; }
float length(Vertex v){ return sqrtf(dot(v, v)); }
Vertex normalize( Vertex v ){ return scale(v, 1.0f / length( v )); }

void save(){
  printf("P3\n %d \t %d\n 255\n", WIDTH, HEIGHT);
  for (int y = 0; y < HEIGHT; y++){
    for (int x = 0; x < WIDTH; x++){
      for (int c = 0; c < 3; c++){
        printf("%d \t", img[x][y][c]);
      }
      printf("\n");
    }
  }
}

// Retorna t (distância) ou -1 se não há interseção
float intersecao_esfera( Vertex O, Vertex D, Vertex C, float r ){
  Vertex L = sub( O, C );
  float a = dot( D, D );
  float b = 2.0 * dot(L, D);
  float c = dot(L, L) - r * r;
  float delta = b * b - 4 * a * c;
  if ( delta < 0 ) return -1.0;
  float sqrt_delta = sqrtf( delta );
  float t0 = (-b - sqrt_delta) / (2 * a);
  float t1 = (-b + sqrt_delta) / (2 * a);
  if ( t0 > 0.001 ) return t0;
  if ( t1 > 0.001 ) return t1;
  return -1.0;
}
```

```
// Retorna t (distância) ou -1 se não há interseção
float intersecao_plano(Vertex O, Vertex D, Vertex P, Vertex N){
  float denominador = dot( N, D );
  if ( fabs(denominador) < 0.0 ) return -1.0;
  float t = dot( sub(P, O), N ) / denominador;
  return ( t > 0.0 ) ? t : -1.0;
}

void render(){
  Vertex camera = (Vertex){0, 0, 0};
  Vertex light_position = normalize( (Vertex){1, 1, 1} );

  // Parâmetros da esfera
  Vertex C = (Vertex){0, 0, -5};
  float r = 1.0;

  // Parâmetros do plano
  Vertex ponto_plano = (Vertex){0, -1, 0};
  Vertex normal_plano = (Vertex){0, 1, 0};


  for (int y = 0; y < HEIGHT; y++){
    for (int x = 0; x < WIDTH; x++){
      float fx = (2.0 * x / WIDTH - 1.0) * (float)WIDTH / HEIGHT;
      float fy = 1.0 - 2.0 * y / (float) HEIGHT;
      Vertex D = normalize( (Vertex){fx, fy, -1} );

      float ts = intersecao_esfera(camera, D, C, r);
      float tp = intersecao_plano(camera, D, ponto_plano, normal_plano);

      if ( ts > 0.0 && (tp < 0 || ts < tp) ){
        Vertex hitPoint = add( camera, scale( D, ts ));
        Vertex normal = normalize( sub(hitPoint, C) );
        float diff = fmaxf(0.0, dot(normal, light_position));
        img[x][y][0] = (unsigned char)(diff * 255);
        img[x][y][1] = 0;
        img[x][y][2] = 0;
      }
      else{
        if ( tp > 0 ){
          Vertex hitPoint = add( camera, scale( D, tp ) );
          // Criar um xadrez
          float xadrez = ((int)(floor(hitPoint.x) + floor(hitPoint.z))) % 2;
          // Quando xadrez for par imprime mais claro e quando for ímpar mais escuro
          unsigned char c = xadrez ? 155 : 80;
          float diff = fmaxf(0.0, dot(normal_plano, light_position));
          img[x][y][0] = c;
          img[x][y][1] = c;
          img[x][y][2] = c;
        }
        else{
          img[x][y][0] = 135;
          img[x][y][1] = 206;
          img[x][y][2] = 250;
        }
      }
    }
  }
}

int main(){
  render();
  save();
  return 0;
}
```

Renderização de uma esfera com plano e reflexão especular

```c
/**
 * \file raytracing3.c
 *
 * \brief Implementação do traçador de raios para criação de imagens
 * com renderização de uma esfera e um plano. Além disso, implementamos
 * a reflexão.
 *
 * \author
 * Petrucio Ricardo Tavares de Medeiros \n
 * Universidade Federal Rural do Semi-Arido \n
 * Departamento de Engenharias e Tecnologia \n
 * petrucio at ufersa (dot) edu (dot) br
 *
 * \version 1.0
 * \date Jul 2025
 */

#include <stdio.h>
#include <math.h>

#define WIDTH 800
#define HEIGHT 600

#define MAX_DEPTH 2

unsigned char img[WIDTH][HEIGHT][3];

typedef struct{
  float x, y, z;
} Vertex;

Vertex add( Vertex a, Vertex b ){ return (Vertex){a.x + b.x, a.y + b.y, a.z + b.z}; }
Vertex sub( Vertex a, Vertex b ){ return (Vertex){a.x - b.x, a.y - b.y, a.z - b.z}; }
Vertex scale( Vertex v, float s ){ return (Vertex){s*v.x, s*v.y, s*v.z}; }
float dot(Vertex a, Vertex b){ return a.x*b.x + a.y*b.y + a.z*b.z; }
float length(Vertex v){ return sqrtf(dot(v, v)); }
Vertex normalize( Vertex v ){ return scale(v, 1.0f / length( v )); }
Vertex reflect(Vertex D, Vertex N){ return sub(D, scale(N, 2.0f * dot(D, N))); }

void save(){
  printf("P3\n %d \t %d\n 255\n", WIDTH, HEIGHT);
  for (int y = 0; y < HEIGHT; y++){
    for (int x = 0; x < WIDTH; x++){
      for (int c = 0; c < 3; c++){
        printf("%d \t", img[x][y][c]);
      }
      printf("\n");
    }
  }
}

// Retorna t (distância) ou -1 se não há interseção
float intersecao_esfera( Vertex O, Vertex D, Vertex C, float r ){
  Vertex L = sub( O, C );
  float a = dot( D, D );
  float b = 2.0 * dot(L, D);
  float c = dot(L, L) - r * r;
  float delta = b * b - 4 * a * c;
  if ( delta < 0 ) return -1.0;
  float sqrt_delta = sqrtf( delta );
  float t0 = (-b - sqrt_delta) / (2 * a);
  float t1 = (-b + sqrt_delta) / (2 * a);
  if ( t0 > 0.001 ) return t0;
  if ( t1 > 0.001 ) return t1;
  return -1.0;
}

// Retorna t (distância) ou -1 se não há interseção
float intersecao_plano(Vertex O, Vertex D, Vertex P, Vertex N){
```

```
    float denominador = dot( N, D );
    if ( fabs(denominador) < 0.0 ) return -1.0;
    float t = dot( sub(P, O), N ) / denominador;
    return ( t > 0.0 ) ? t : -1.0;
  }

  Vertex trace(Vertex O, Vertex D, int depth){
    Vertex light_position = normalize( (Vertex){1, 1, 1} );

    // Parâmetros da esfera
    Vertex C = (Vertex){0, 0, -5};
    float r = 1.0;

    // Parâmetros do plano
    Vertex ponto_plano = (Vertex){0, -1, 0};
    Vertex normal_plano = (Vertex){0, 1, 0};

    float ts = intersecao_esfera(O, D, C, r);
    float tp = intersecao_plano(O, D, ponto_plano, normal_plano);

    if ( ts > 0.0 && (tp < 0 || ts < tp) ){
      Vertex hitPoint = add( O, scale( D, ts ));
      Vertex normal = normalize( sub(hitPoint, C) );
      float diff = fmaxf(0.0, dot(normal, light_position));

      // Cor difusa
      Vertex cor = scale((Vertex){255, 0, 0}, diff);

      // Reflexão
      if ( depth < MAX_DEPTH ){
        Vertex refl_dir = normalize(reflect(D, normal));
        Vertex refl_color = trace(add(hitPoint, scale(normal, 1e-4)), refl_dir, depth + 1);
        cor = add(scale(cor, 0.5f), scale(refl_color, 0.5f));
      }
      return cor;
    }
    else{
      if ( tp > 0 ){
        Vertex hitPoint = add( O, scale( D, tp ) );
        // Criar um xadrez
        float xadrez = ((int)(floor(hitPoint.x) + floor(hitPoint.z))) % 2;
        // Quando xadrez for par imprime mais claro e quando for ímpar mais escuro
        unsigned char c = xadrez ? 155 : 80;
        float diff = fmaxf(0.0, dot(normal_plano, light_position));
        return (Vertex){c, c, c};
      }
      else{
        return (Vertex){135, 206, 250}; // fundo azul claro
      }
    }
  }

  void render(){
    Vertex camera = (Vertex){0, 0, 0};

    for (int y = 0; y < HEIGHT; y++){
      for (int x = 0; x < WIDTH; x++){
        float fx = (2.0 * x / WIDTH - 1.0) * (float)WIDTH / HEIGHT;
        float fy = 1.0 - 2.0 * y / (float) HEIGHT;
        Vertex D = normalize( (Vertex){fx, fy, -1} );
        Vertex color = trace(camera, D, 0);
        img[x][y][0] = (unsigned char)color.x;
        img[x][y][1] = (unsigned char)color.y;
        img[x][y][2] = (unsigned char)color.z;
      }
    }
  }

  int main(){
```

```
    render();
    save();
    return 0;
}
```

Last updated 2025-07-23 06:02:35 -03