DCC192

2025/1

UFMG

Desenvolvimento de Jogos Digitais

A28: Gráficos 3D — Câmeras e Projeções

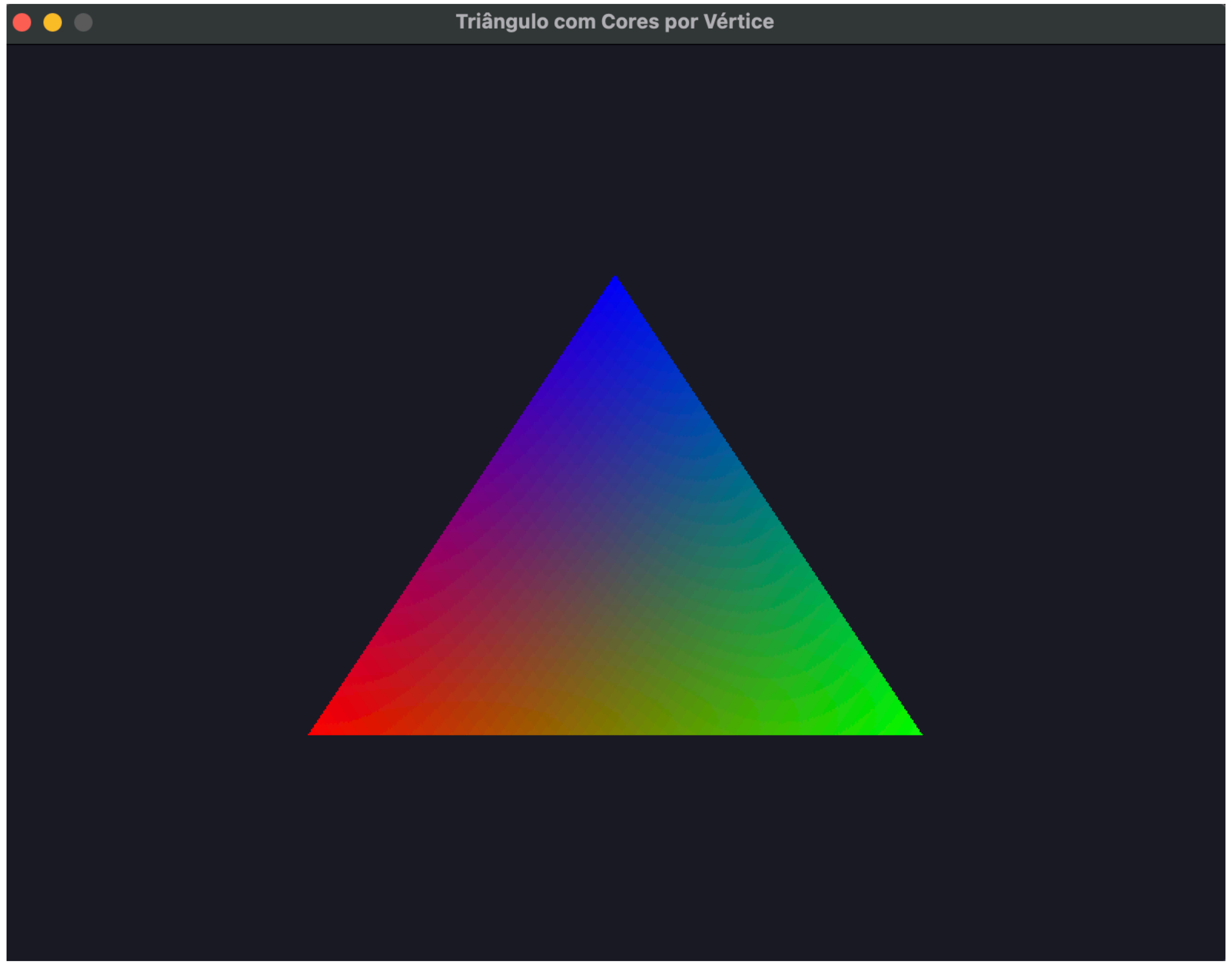
Prof. Lucas N. Ferreira

Plano de Aula

- Cores
- Compondo objetos com triângulos
- Câmeras
- Projeções
  - Perspectiva
  - Ortográfica

Atributos dos Vértices

Até agora, vimos como criar e transformar triângulos usando transformações geométricas. Mas ainda não vimos como passar atributos para os vértices nos shaders.

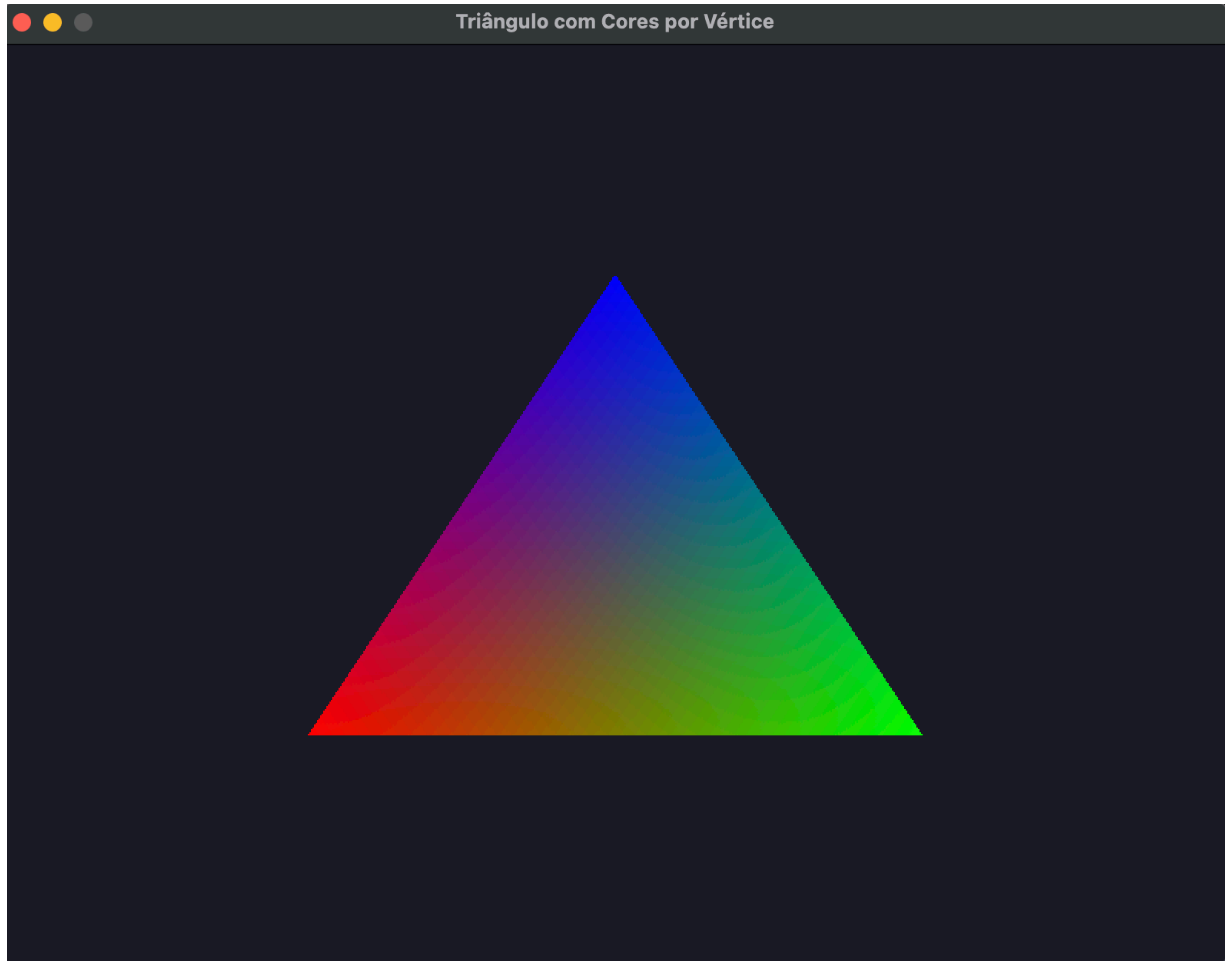


Primeiramente, temos que passar as cores de cada vértice como atributos, intercalando-os com os valores de posição

```
float vertices[] = {
//      x       y        r     g     b
     -0.5f, -0.5f,  1.0f, 0.0f, 0.0f, // V1: vermelho
      0.5f, -0.5f,  0.0f, 1.0f, 0.0f, // V2: verde
      0.0f,  0.5f,  0.0f, 0.0f, 1.0f  // V3: azul
 };
```

Atributos dos Vértices

Até agora, vimos como criar e transformar triângulos usando transformações geométricas. Mas ainda não vimos como passar atributos para os vértices nos shaders.

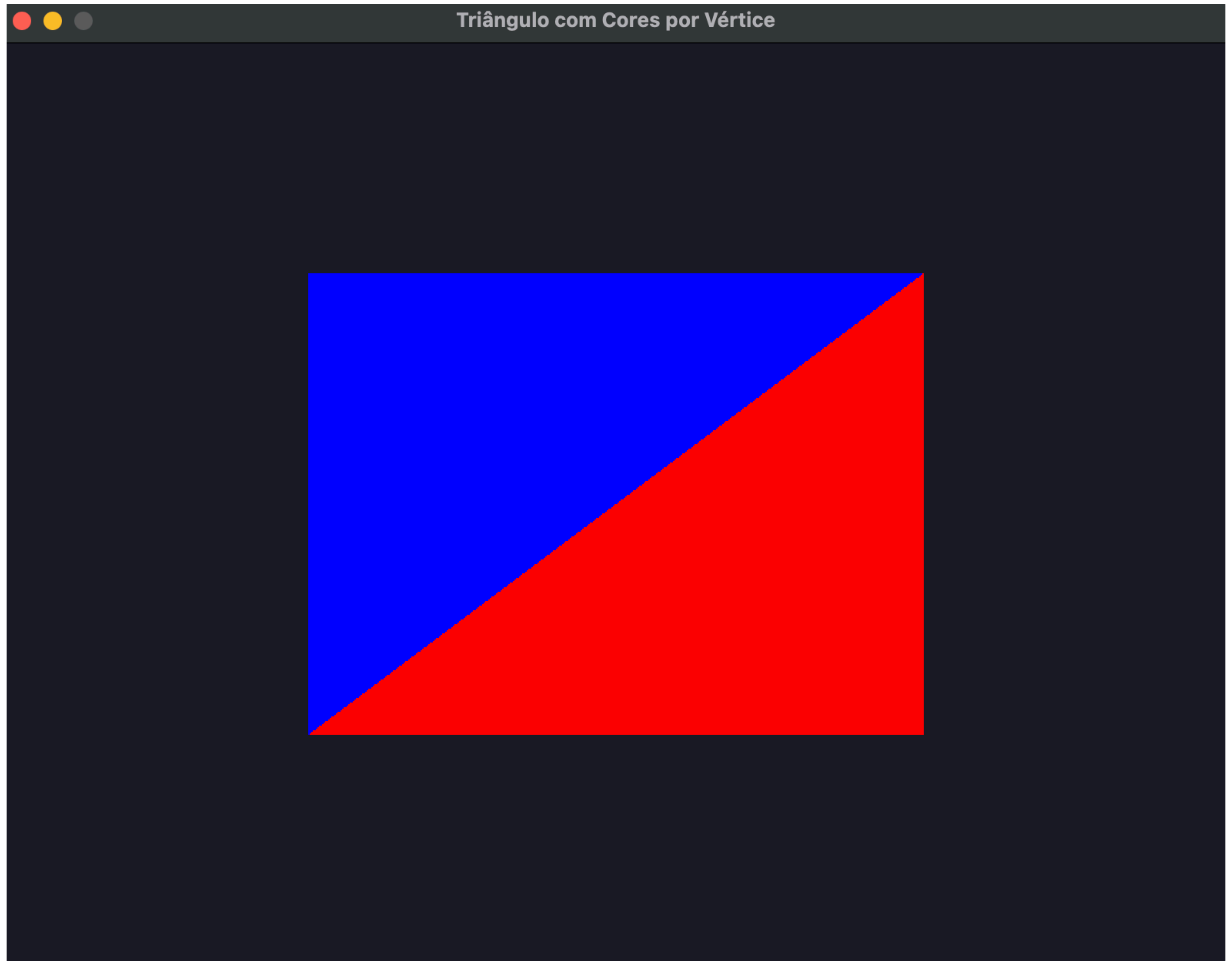


Em seguida, temos que criar variáveis out/in nos shaders para passar as cores até o Fragment Shader:

```
constexpr std::string_view vertexShaderSource = R"(
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;

out vec3 vertexColor;

void main() {
    gl_Position = vec4(aPos, 0.0, 1.0);
    vertexColor = aColor;
})";

constexpr std::string_view fragmentShaderSource = R"(
#version 330 core
in vec3 vertexColor;
out vec4 FragColor;

void main() {
    FragColor = vec4(vertexColor, 1.0);
})”;
```

Modelos mais Complexos

Também não vimos como combinar triângulos para formar modelos mais complexos. Vejamos um exemplo simples: criando um retângulo a partir de triângulos:

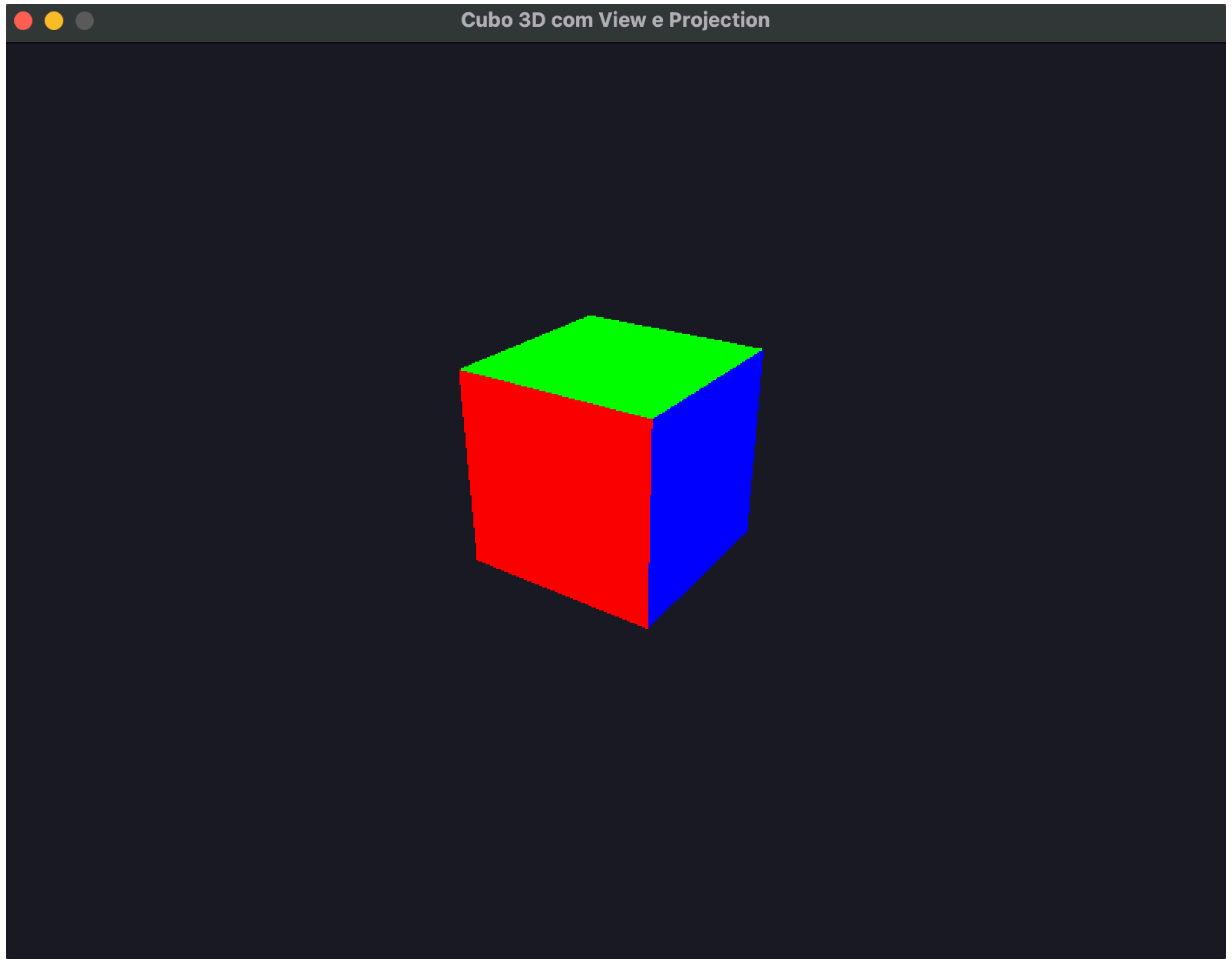


Em seguida, temos que criar variáveis out/in nos shaders para passar as cores até o Fragment Shader:

```
// Posição (x, y) e Cor (r, g, b) para cada vértice
float vertices[] = {
    // Triângulo 1 - azul
    -0.5f, -0.5f,   0.0, 0.0, 1.0,  // Inferior esquerdo
    -0.5f,  0.5f,   0.0, 0.0, 1.0,  // Superior esquerdo
     0.5f,  0.5f,   0.0, 0.0, 1.0,  // Superior direito

    // Triângulo 2 - amarelo
    -0.5f, -0.5f,   1.0f, 0.0f, 0.0f,  // Inferior esquerdo (duplicado)
     0.5f,  0.5f,   1.0f, 0.0f, 0.0f,  // Superior direito (duplicado)
     0.5f, -0.5f,   1.0f, 0.0f, 0.0f   // Inferior direito
};
```

Câmera

Agora que sabemos como criar modelos mais complexos usando triângulos com cores diferentes, vamos entender como adicionar uma câmera em cenas 3D:



Câmeras são tipicamente implementadas com duas transformações extras:

1. Transformação da câmera

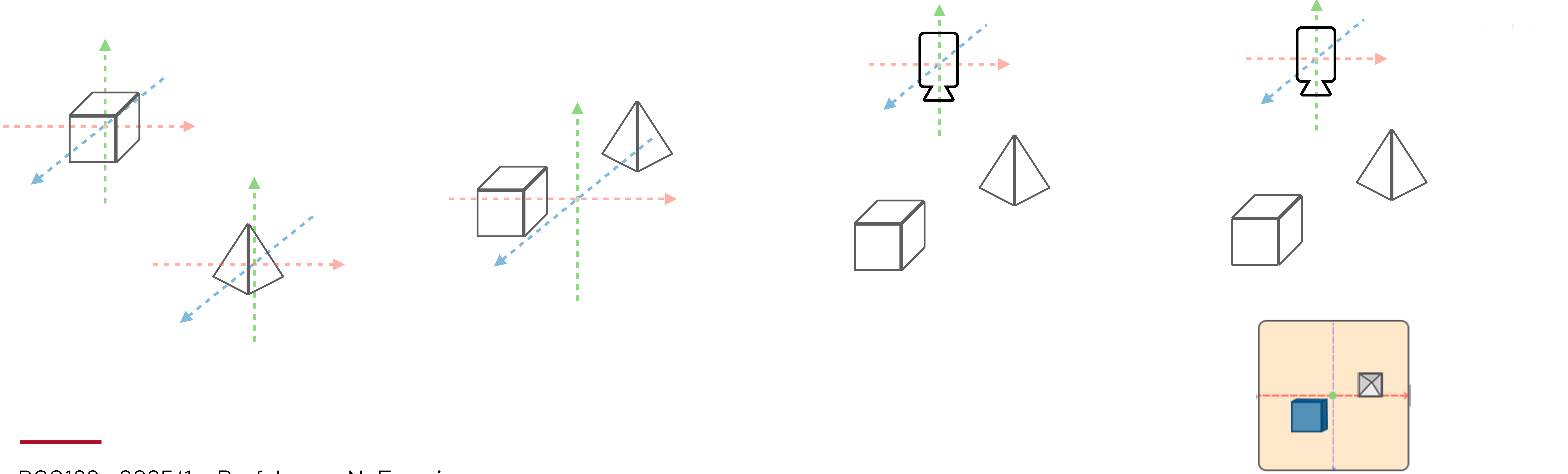
2. Projeção perspectiva

Ambas essas transformações também são representadas por matrizes 4×4

Sistemas de Coordenadas

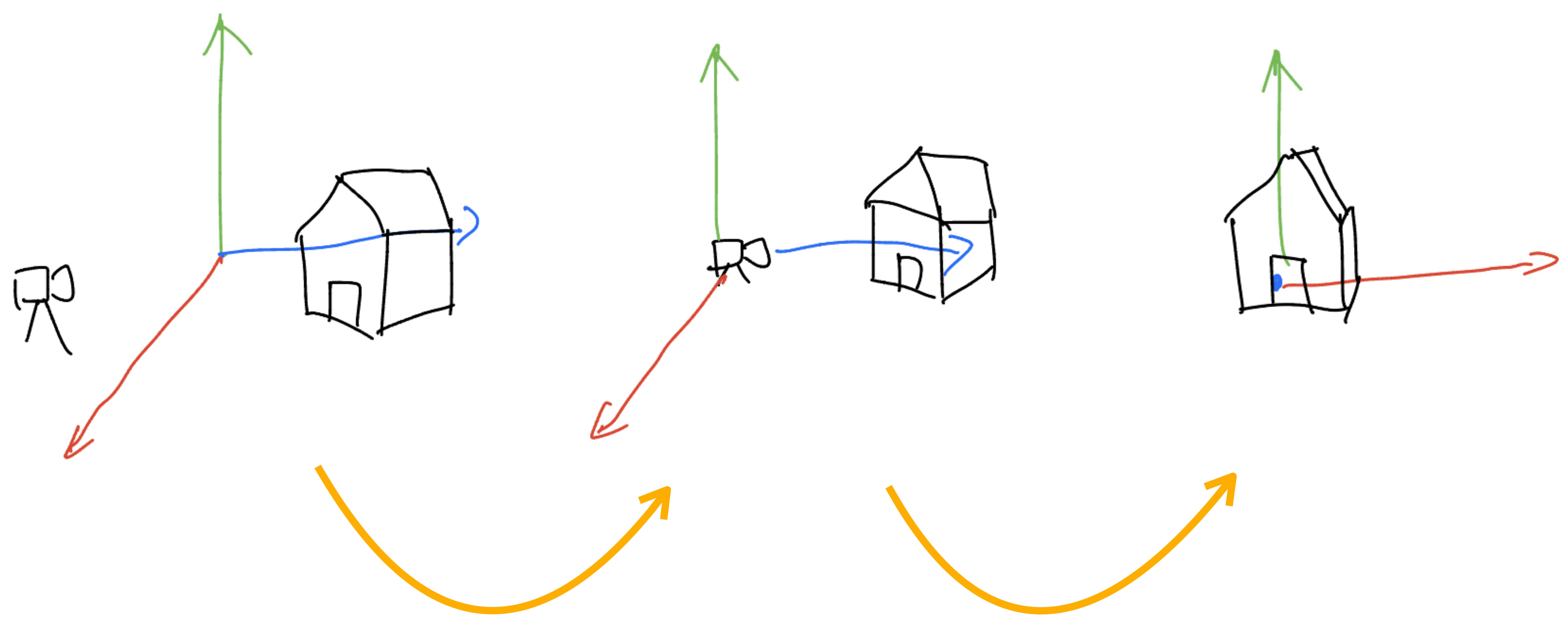
Antes de entender como criar as matrizes de transformação e projeção de câmeras. Vamos entender o fluxo geral de transformações que ocorrem nos vértices de entrada:

3D	Matriz	3D	Matriz	3D	Matriz	2D	Matriz
1. Sist. de Coords. do Objeto	→ Modelo	2. Sist. de Coords. do Mundo	→ View	3. Sist. de Coords. da Câmera	→ Projeção	4. Sist. de Coords. Clip	→ Viewport



Sistema de Coordenadas da Câmera

A matriz de transformação de câmera mapeia os vértices do sistemas de coordenadas do mundo para o sistema de coordenadas da câmera:



Rotação

$$= \begin{bmatrix} L_x & U_x & F_x & 0 \\ L_y & U_y & F_y & 0 \\ L_z & U_z & F_z & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

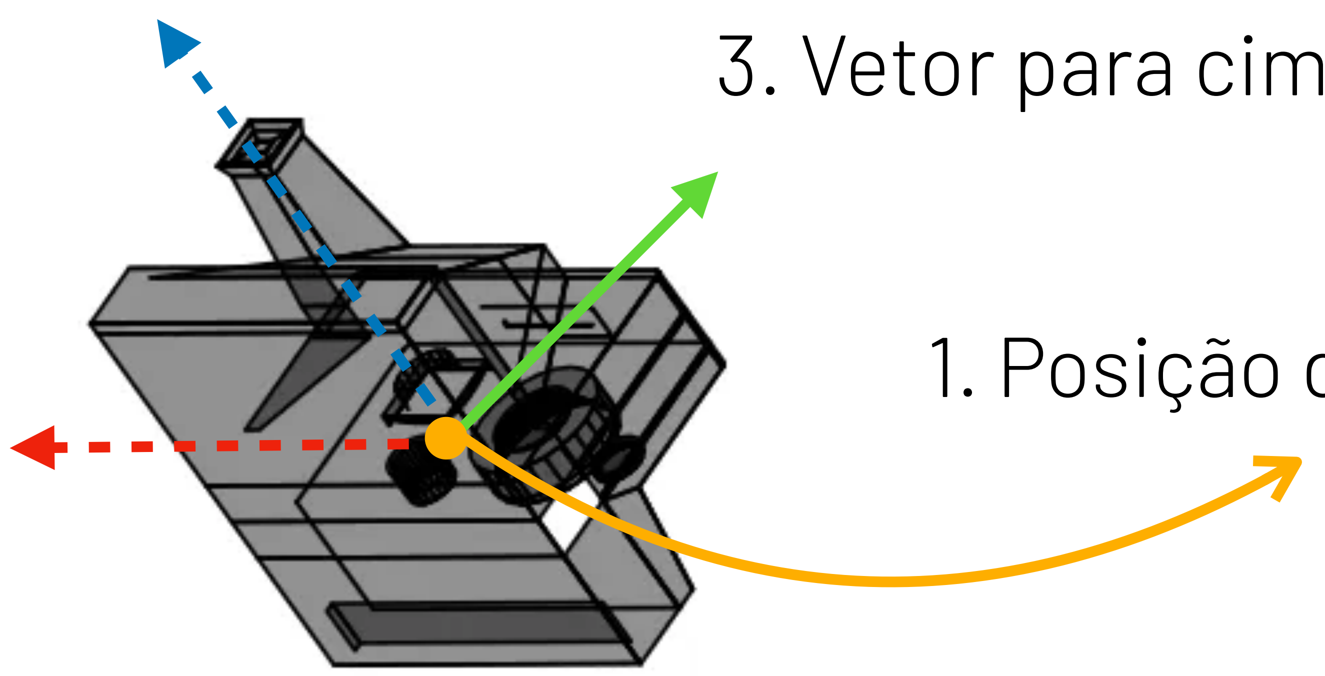
Translação

- L é o *Left Vector*
- U é o *Up Vector*
- F é o *Forward Vector*
- T *é um vetor de translação*

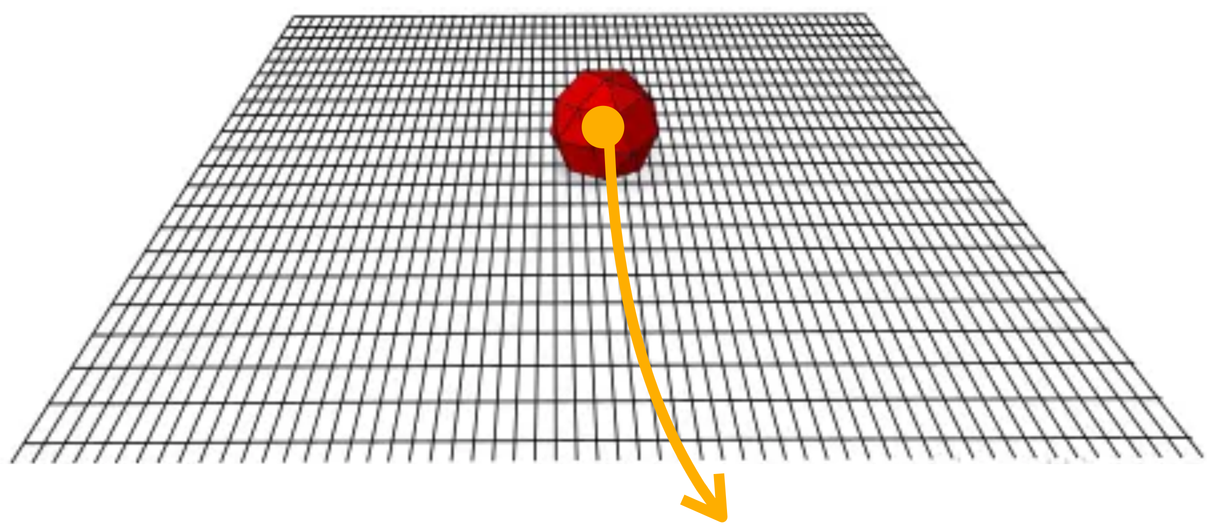
1. Translação para que a câmera fique no centro
2. Rotação para alinhar com o sistema da câmera

Matriz de transformação de câmera

A matriz de transformação de câmera é criada a partir **(1) da posição da câmera**, **(2), de um ponto de olhar e (3) um vetor "para cima" da câmera**:



$$\begin{bmatrix} L_x & U_x & F_x & 0 \\ L_y & U_y & F_y & 0 \\ L_z & U_z & F_z & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

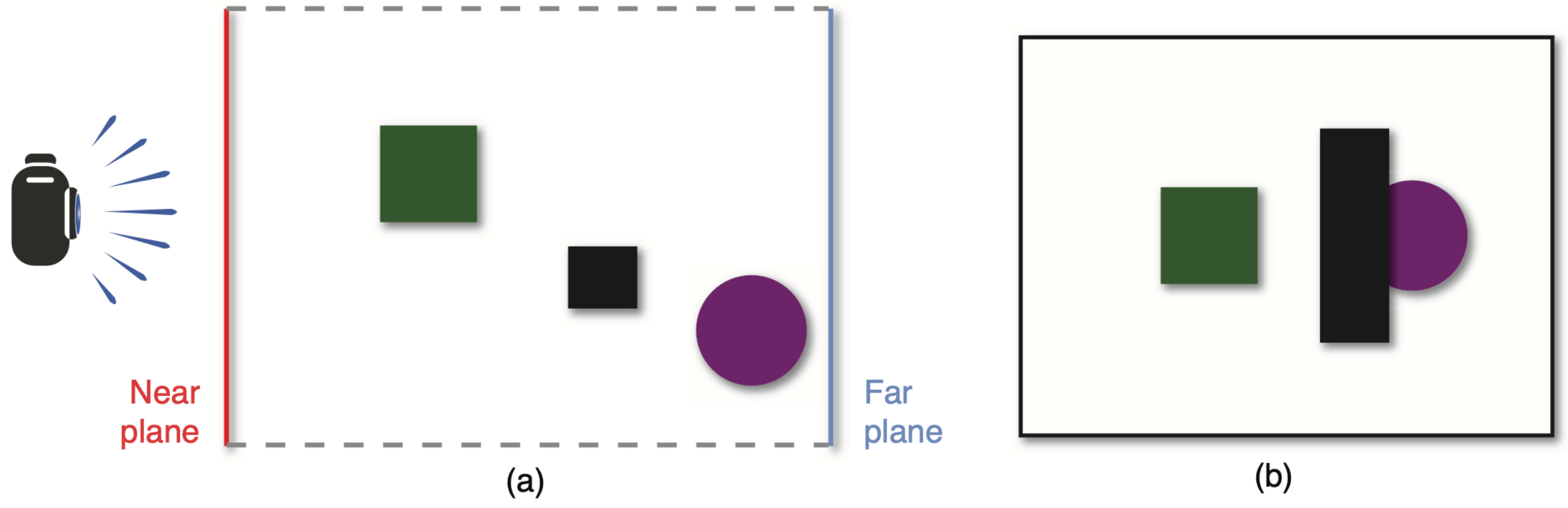



```
Matrix4 CreateLookAt(Vector3& eye, Vector3& target, Vector3& up)
{
    Vector3 zaxis = Vector3::Norm(target - eye);
    Vector3 xaxis = Vector3::Norm(Vector3::Cross(up, zaxis));
    Vector3 yaxis = Vector3::Norm(Vector3::Cross(zaxis, xaxis));
    Vector3 trans;
    trans.x = -Vector3::Dot(xaxis, eye);
    trans.y = -Vector3::Dot(yaxis, eye);
    trans.z = -Vector3::Dot(zaxis, eye);

    float temp[4][4] =
    {
        { xaxis.x, yaxis.x, zaxis.x, 0.0f },
        { xaxis.y, yaxis.y, zaxis.y, 0.0f },
        { xaxis.z, yaxis.z, zaxis.z, 0.0f },
        { trans.x, trans.y, trans.z, 1.0f }
    };
    return Matrix4(temp);
}
```

Projeções

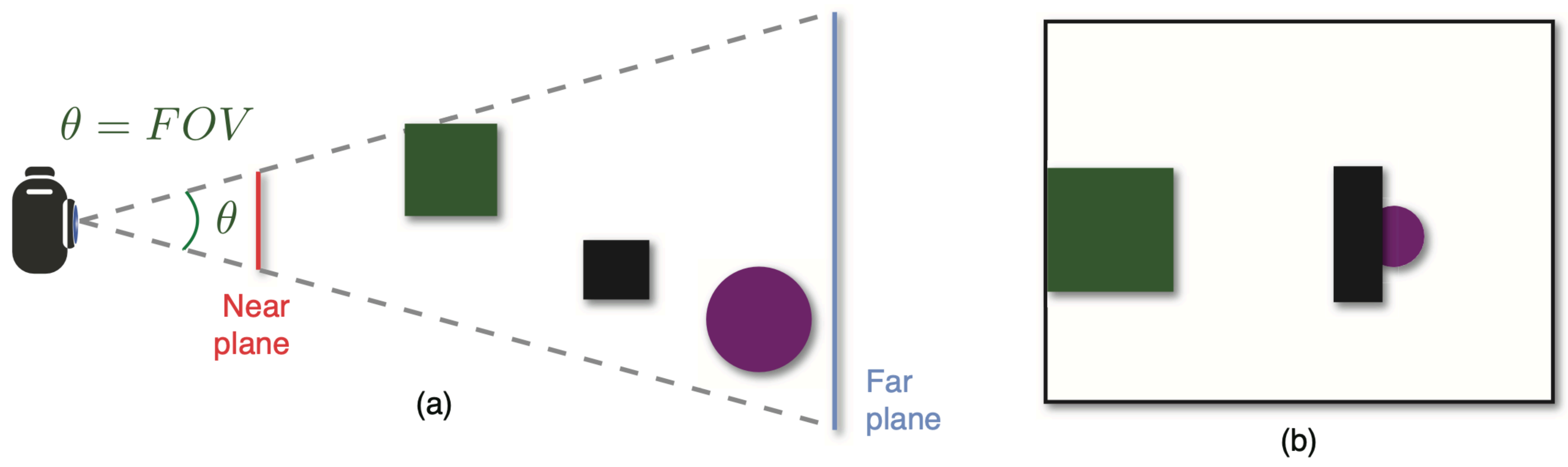
Uma **projeção** transforma pontos de uma dimensão n em uma dimensão $m < n$. Em computação gráfica, as projeções mais interessam são do $\mathbb{R}^3 \rightarrow \mathbb{R}^2$ ou $(x, y, z) \rightarrow (x, y)$

Projeção Ortográfica



Objetos mais distantes da câmera têm o mesmo tamanho que objetos mais próximos dela.

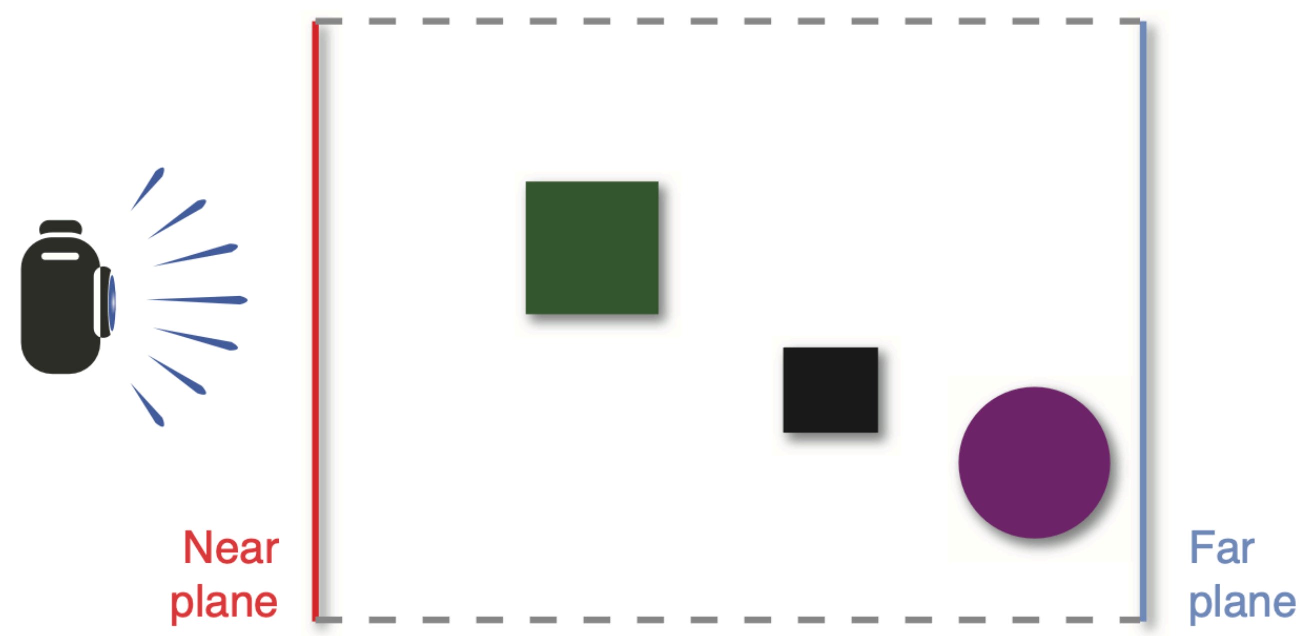
Projeção Perspectiva

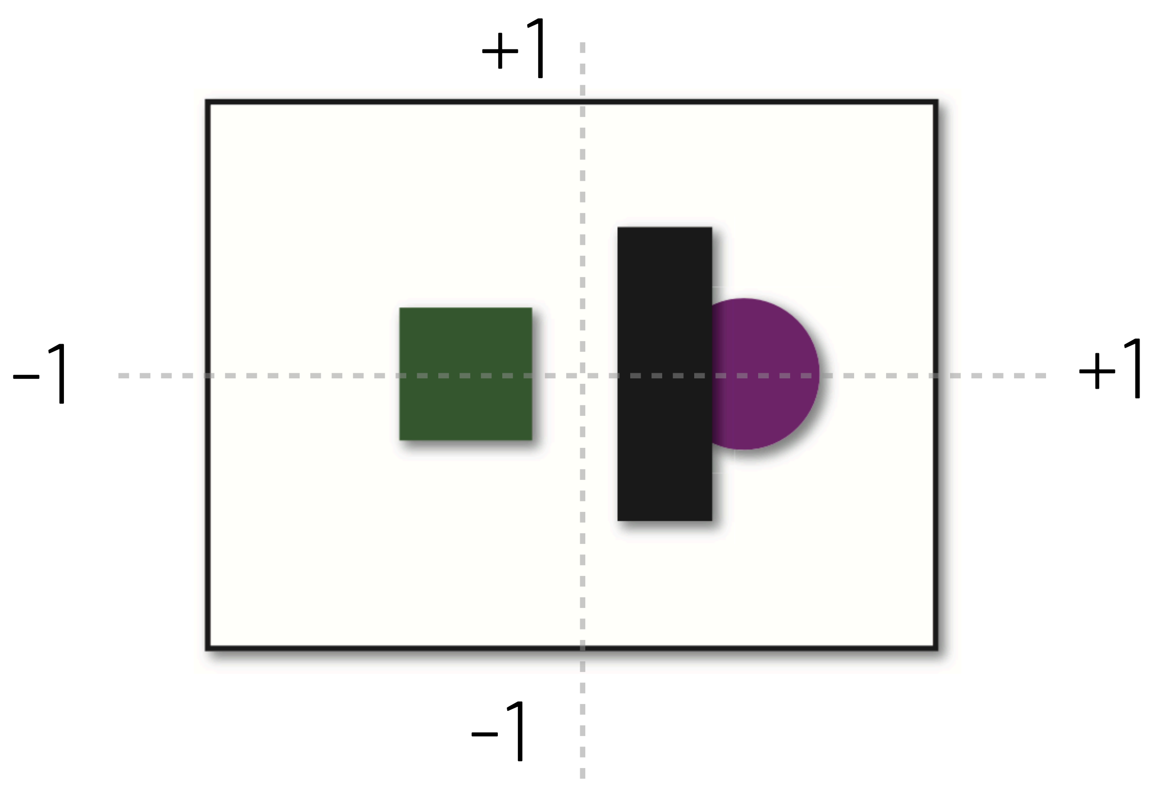


Objetos mais distantes da câmera ficam menores do que os mais próximos.

Projeção Ortográfica

Em jogos 2D, a projeção mais comum é a **Projeção Ortográfica,** onde o mundo é restrito por uma AABB (3D) e mapeado para um espaço de recorte (imagem 2D).





A matriz de projeção ortográfica é composta por quatro parâmetros:

- `width` e `height`: largura e a altura da telaolho
- `Near` e `Far`: planos de corte próximos e distantes (em profundidade)

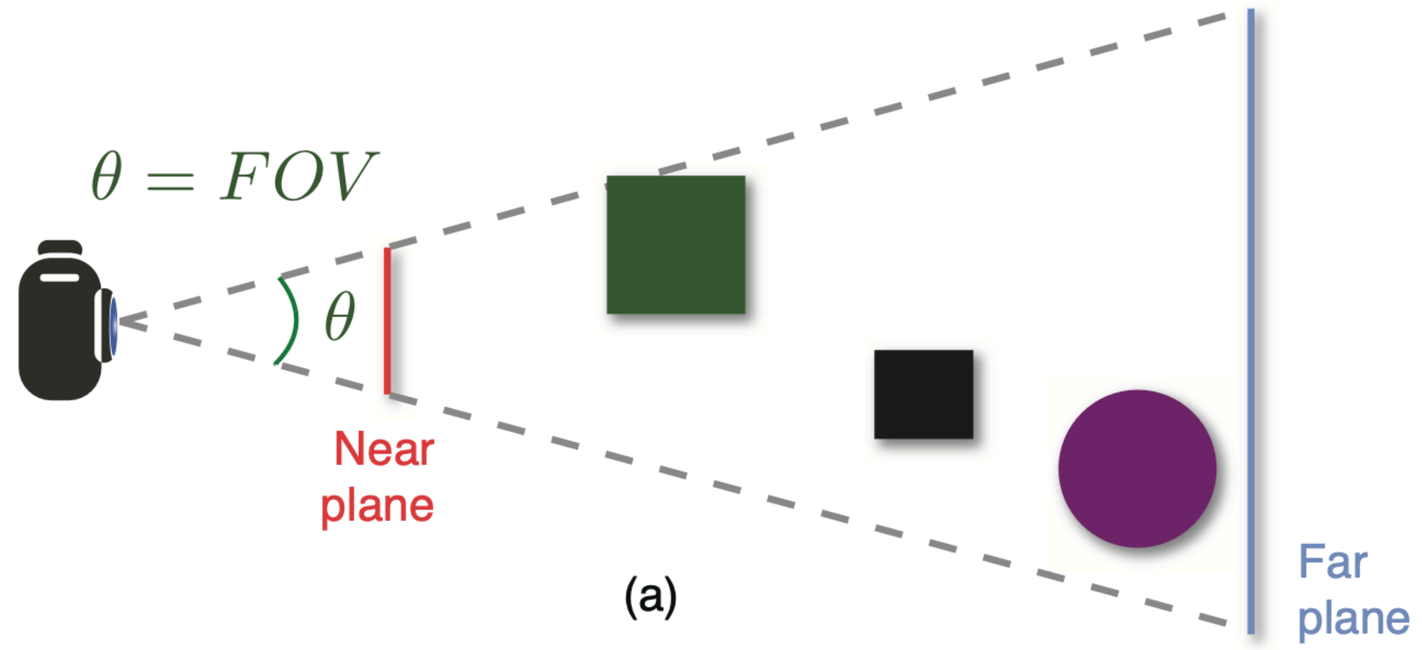
$$Orthographic = \begin{bmatrix} \frac{2}{width} & 0 & 0 & 0 \\ 0 & \frac{2}{height} & 0 & 0 \\ 0 & 0 & \frac{1}{far - near} & 0 \\ 0 & 0 & \frac{near}{near - far} & 1 \end{bmatrix}$$

Escala: mapeia objetos para intervalo de dimensão [-1,1]

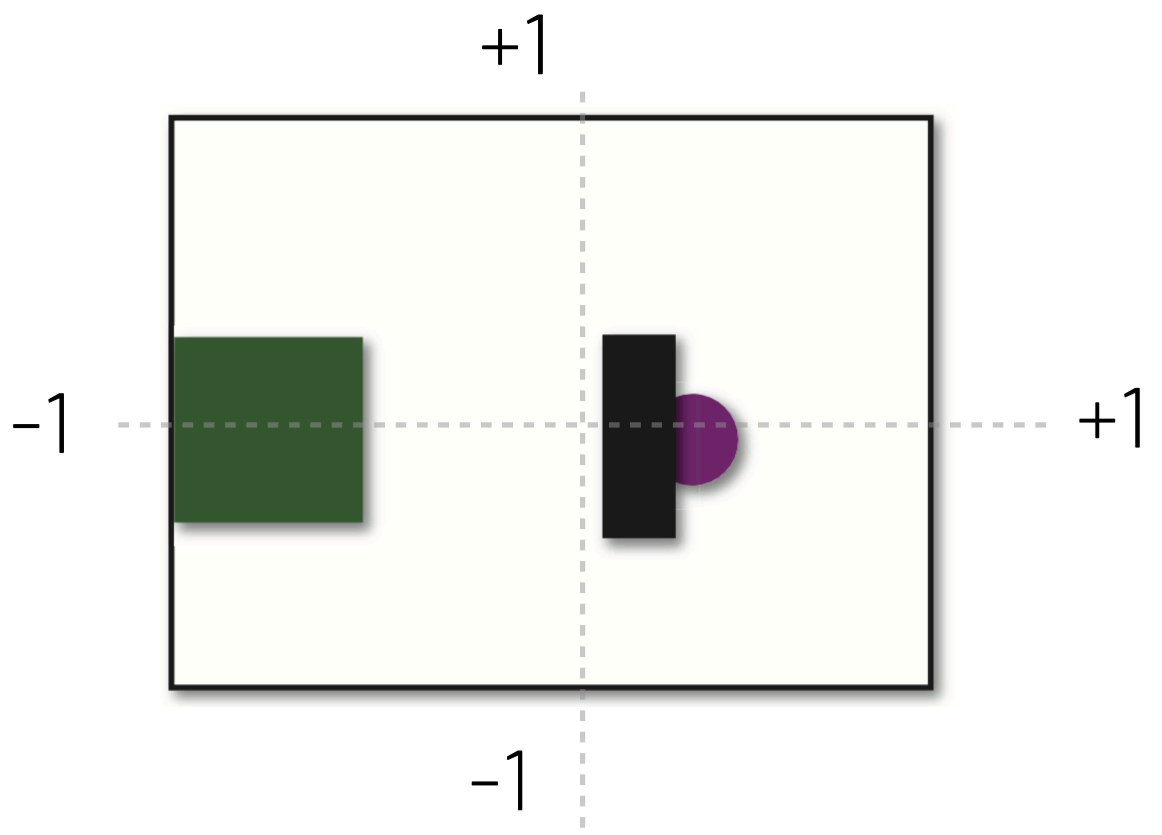
Translação: move os valores de profundidade para o intervalo [-1, 1] no eixo z

Projeção Perspectiva

Em jogos 3D, a projeção mais comum é a **Projeção Perspectiva,** onde o mundo é restrito por uma Pirâmide (3D) e mapeado para um espaço de recorte (imagem 2D).



(a)



A matriz de projeção perspectiva é composta por cinco parâmetros:

- `Width` e `Height`: largura e a altura da telaolho
- `Near` e `Far`: planos de corte próximos e distantes (em profundidade)
- `FOV`: ângulo ao redor da câmera que é visível na projeção

$$yScale = \cot(fov / 2)$$

$$xScale = yScale \cdot \frac{height}{width}$$

$$Perspective = \begin{bmatrix} xScale & 0 & 0 & 0 \\ 0 & yScale & 0 & 0 \\ 0 & 0 & \dfrac{far}{far - near} & 1 \\ 0 & 0 & \dfrac{-near \cdot far}{far - near} & 0 \end{bmatrix}$$

Próxima aula

A29: Shaders

- Mapeamento de Texturas
- Modelos de Iluminação
    - Tipos de luz
    - Modelos de Phong
    - Ray Tracing