UFABC

Computação Gráfica

André Brandão

Aula 16

Aula prática

Cubo colorido

Aula 15

- Trabalhamos com matrizes de transformação

Cubo colorido

- Nesta aula, trabalhamos com a inserção de um cubo colorido, na janela.
- Um cubo tem seis faces quadradas. Uma vez que OpenGL só sabe sobre triângulos, teremos que desenhar 12 triângulos: dois para cada face.
- Nós apenas definimos nossos vértices da mesma maneira que fizemos para o triângulo.

Definição de vértices

- Os vértices são definidos por três valores do tipo float.
- Um cubo é composto por seis vértices, cada um com dois triângulos.

Definição de vértices

```
// Our vertices. Three consecutive floats give a 3D
vertex; Three consecutive vertices give a triangle.
// A cube has 6 faces with 2 triangles each, so this
makes 6*2=12 triangles, and 12*3 vertices
static const GLfloat g_vertex_buffer_data[] = {
-1.0f,-1.0f,-1.0f, // triangle 1 : begin
-1.0f,-1.0f, 1.0f,
-1.0f, 1.0f, 1.0f, // triangle 1 : end
...
};
```

Desenho dos triângulos

- O buffer do OpenGL é criado, com seus devidos limites, preenchido e configurado com as funções padrão (*glGenBuffers, glBindBuffer, glBufferData, glVertexAttribPointer*).
- A chamada da função *draw* acontece e você só tem que definir o número certo de vértices que devem ser desenhados.

Desenho de triângulos

```
// Draw the triangle!
glDrawArrays(GL_TRIANGLES, 0, 12*3);
// 12*3 indices starting at 0 -> 12 triangles -> 6 squares
```

Adição de cores

- Uma cor é, o mesmo que uma posição: é apenas um dado. Em termos OpenGL, eles são "atributos".
- Vamos adicionar outro atributo.
- Declare suas cores: os três valores RGB por vértice. No exemplo da aula de hoje, as cores foram geradas aleatoriamente.

Adição de cores

```
// One color for each vertex. They were generated randomly.
static const GLfloat g_color_buffer_data[] = {
    0.583f,  0.771f,  0.014f,
    0.609f,  0.115f,  0.436f,
    ...
};
```

Adição de cores

- O buffer é criado, com seus devidos limites e preenchido.

```
GLuint colorbuffer;
glGenBuffers(1, &colorbuffer);
glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(g_color_buffer_data), g_color_buffer_data, GL_STATIC_DRAW);
```

Adição de cores

- A configuração das cores deve ser realizada.

```
// 2nd attribute buffer : colors
glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
glVertexAttribPointer(
    1,                                // attribute. No particular reason for 1, but must match the layout
in the shader.
    3,                                // size
    GL_FLOAT,                         // type
    GL_FALSE,                         // normalized?
    0,                                // stride
    (void*)0                          // array buffer offset
);
```

Adição de cores

- Assim, no *vertex shader*, temos acesso a este buffer adicional:

```
// Notice that the "1" here equals the "1" in glVertexAttribPointer
layout(location = 1) in vec3 vertexColor;
```

Adição de cores

- Agora, nós enviamos o vetor de três posições *vertexColor* ao Fragment Shader.

```
// Output data ; will be interpolated for each fragment.
out vec3 fragmentColor;

void main(){

    [...]

    // The color of each vertex will be interpolated
    // to produce the color of each fragment
    fragmentColor = vertexColor;
}
```

Adição de cores

- No *Fragment Shader*, declaramos, novamente, o *fragmentColor*:

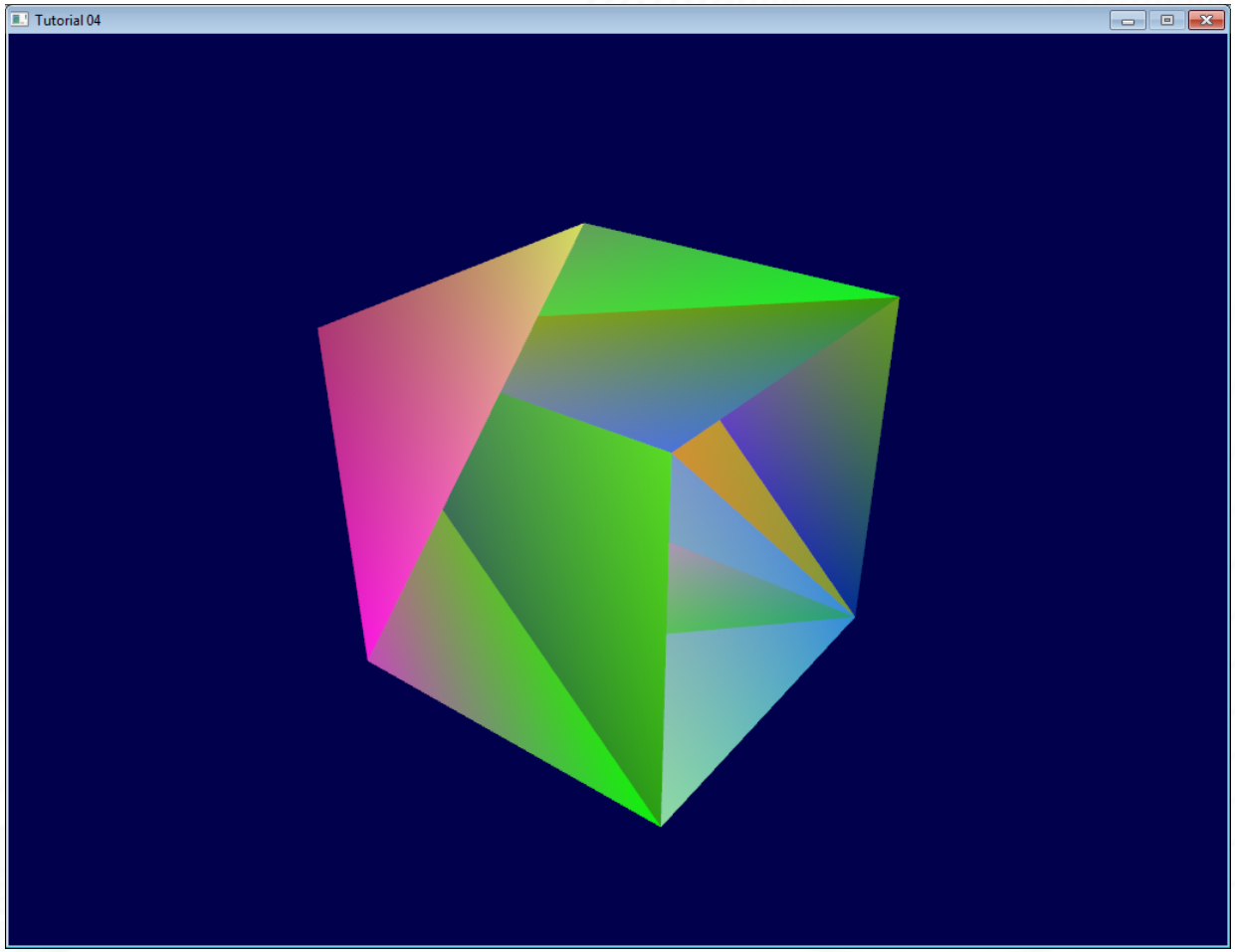
```
// Interpolated values from the vertex shaders
in vec3 fragmentColor;
```

- E copiamos o *fragmentColor* para a cor que será exibida na tela.

Adição de cores

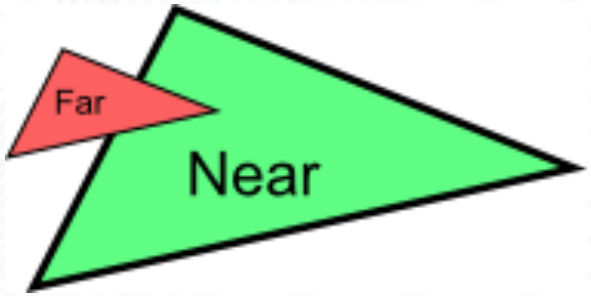
```
// Ouput data
out vec3 color;

void main(){
    // Output color = color specified in the vertex shader,
    // interpolated between all 3 surrounding vertices
    color = fragmentColor;
}
```

Resultado sem Z-Buffer



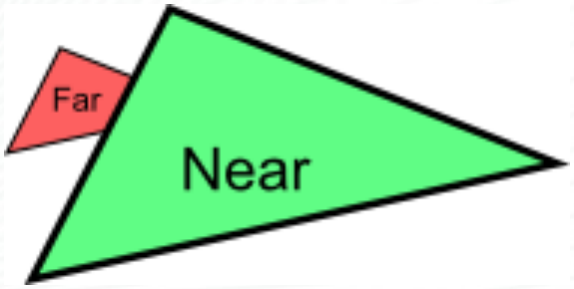
Resultado sem Z-Buffer

- A imagem não ficou clara, pois houve problemas na ordem em que os triângulos do cubo foram renderizados.



Resultado sem Z-Buffer

- Não houve interpretação de profundidade, portanto, triângulos que deveriam estar longe puderam ser renderizados “sobre” triângulos que estão próximos do plano de projeção.
- Desejamos:



Z-Buffer

- A solução para este problema é armazenar o componente de profundidade (ou seja, "Z") de cada fragmento em um buffer.
- Cada vez que desejamos desenhar um fragmento, primeiro, verifica-se se este deve ser desenhado no dado momento (ou seja, o novo fragmento está mais perto do que o anterior).

Z-Buffer

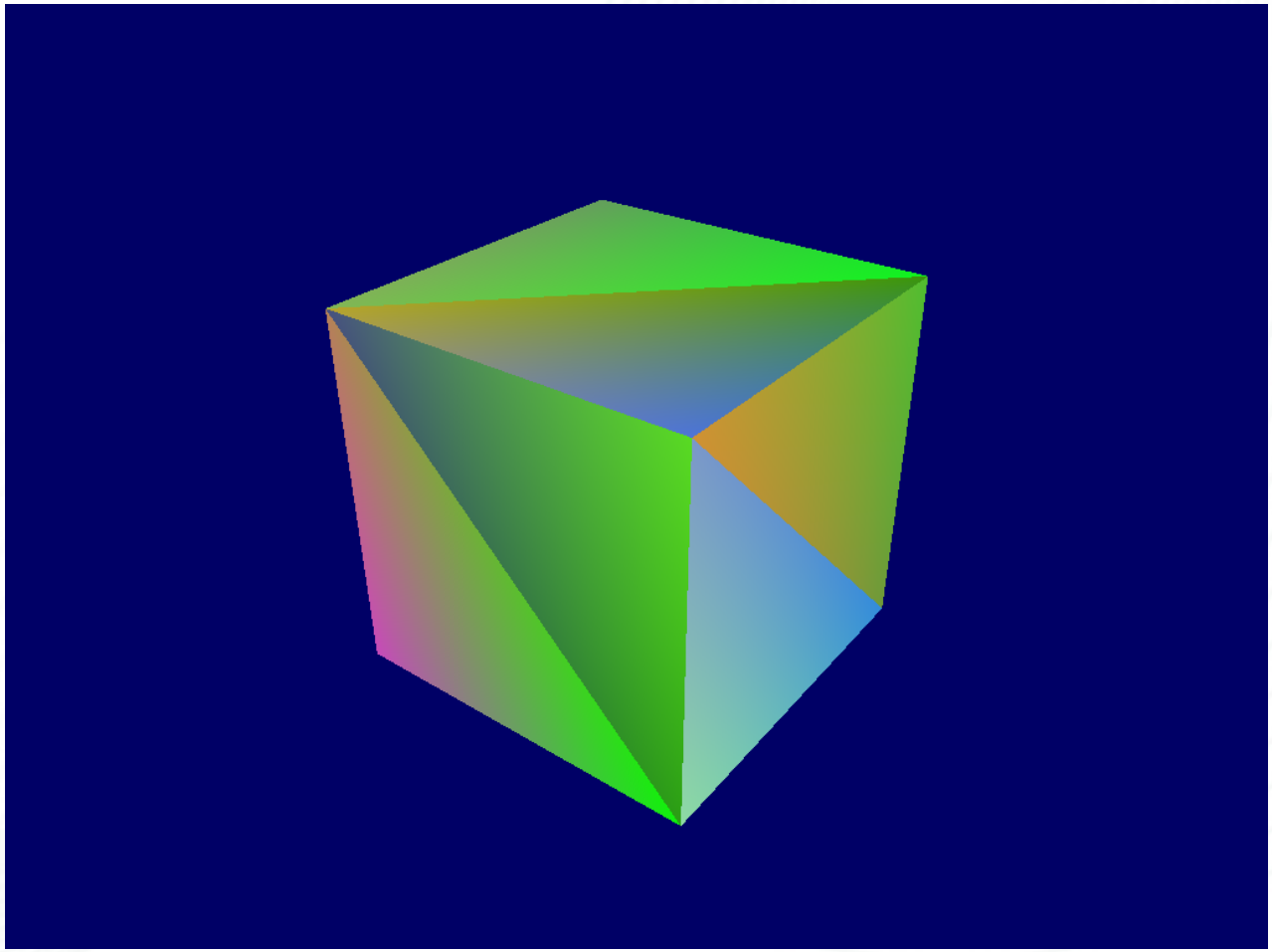
```
// Enable depth test
glEnable(GL_DEPTH_TEST);
// Accept fragment if it closer to the camera than the former one
glDepthFunc(GL_LESS);
```

Z-Buffer

- Você também precisa limpar a profundidade de cada iteração do loop principal.

```
// Clear the screen
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Resultado final



Aula 16

EXERCÍCIOS

Baixe os arquivos do Tidia

- Faça o projeto que apresente o resultado final, ilustrado no slide 23.

Exercícios

- Verifique o código e cada parte do conteúdo abordado nesta aula.
- Não será necessário entregar qualquer atividade relacionada a esta parte do conteúdo.
- Porém, poderá ser cobrada alguma implementação na prova que aborde o conteúdo da aula de hoje.

Referências

- OpenGL Tutorial

  http://www.opengl-tutorial.org/beginners-tutorials/

Fim da Aula 16

André Luiz Brandão