UFABC

Computação Gráfica

André Brandão

Aula 02

Matemática e

OpenGL

Sumário

- Parte 1
  - Continuação da revisão matemática
- Parte 2
  - OpenGL

Revisão matemática

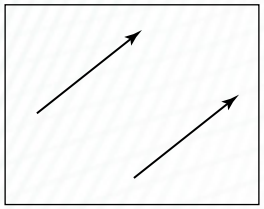
- ~~Conjuntos e logaritmos~~
- ~~Equações de segundo grau~~
- ~~Trigonometria~~
- ~~Ponto~~
- ~~Reta~~
- ~~Comprimento de reta~~
- Vetores

Vetores

- Operações
- Coordenadas cartesianas de um vetor 2D
- Produto escalar
- Produto vetorial
- Bases ortogonais e ortonormais
- Construção de bases

Vetores

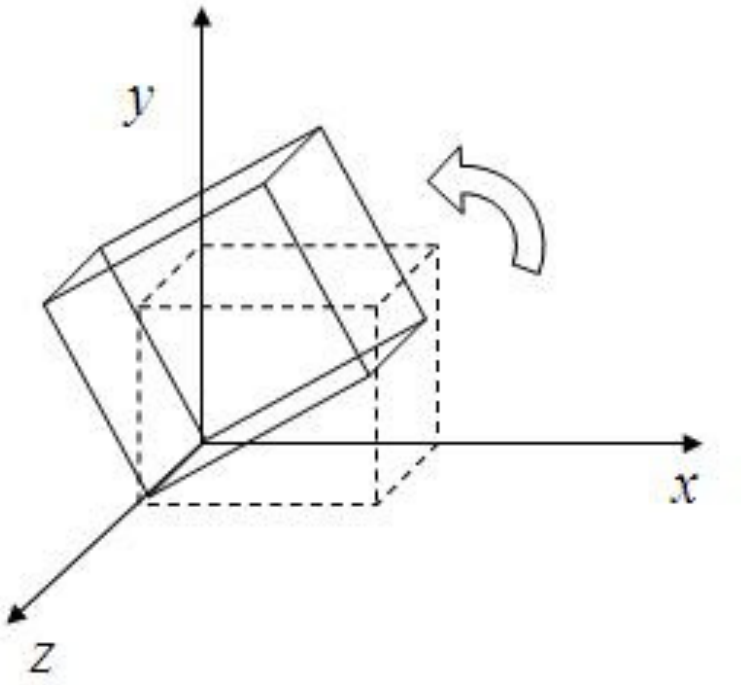
- Vetores tem módulo e direção. Dois vetores são iguais se eles têm o mesmo módulo e direção (Figura).



- Vetores podem ser representados por letras, como, **a**. O comprimento, ou módulo, é representado por $||\mathbf{a}||$.
- Vetores unitários são aqueles que têm comprimento igual a 1.

Vetores

- O vetor zero é o vetor que tem comprimento igual a zero.
- O estudo de vetores é pertinente na Computação Gráfica. Por exemplo, ao deslocar um objeto, cada vértice do mesmo deve ser deslocado.

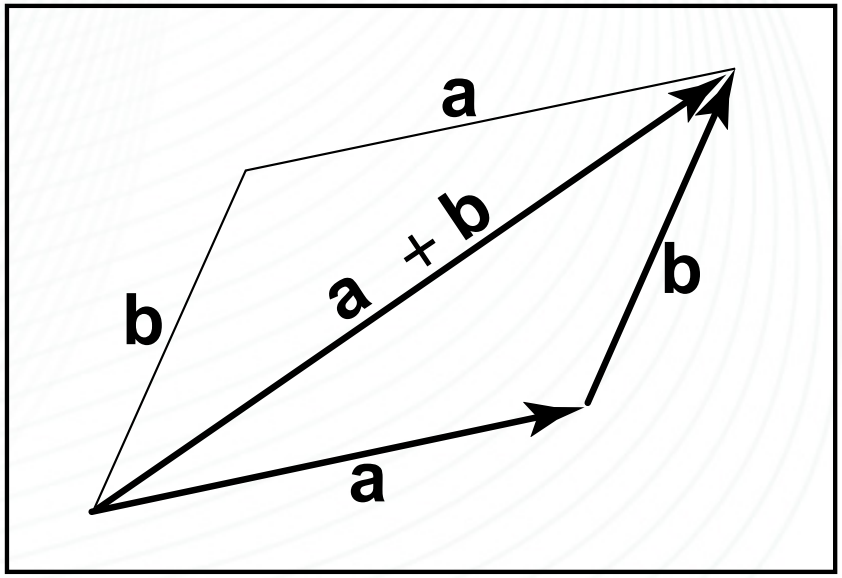


http://cleiton-luiz.blogspot.com.br/

Operações

- Soma: realizada pela regra do paralelogramo.
- O início de um vetor é conectado ao fim do outro vetor. O vetor resultante é o vetor soma (**a + b**).
- A soma é comutativa, ou seja:

$$\mathbf{a} + \mathbf{b} = \mathbf{b} + \mathbf{a}$$



Operações

- Subtração: ao inverter a direção de um vetor **a**, teremos o vetor **–a**. Desta forma, temos a subtração.

$$b - a = - a + b$$

- Multiplicação por uma constante: se uma constante multiplica um vetor, sem mudar a direção deste, então apenas o tamanho do vetor é alterado. Por exemplo, em **k * a**, multiplica-se cada coordenada de **a** por **k**.

Coordenadas cartesianas de um vetor 2D

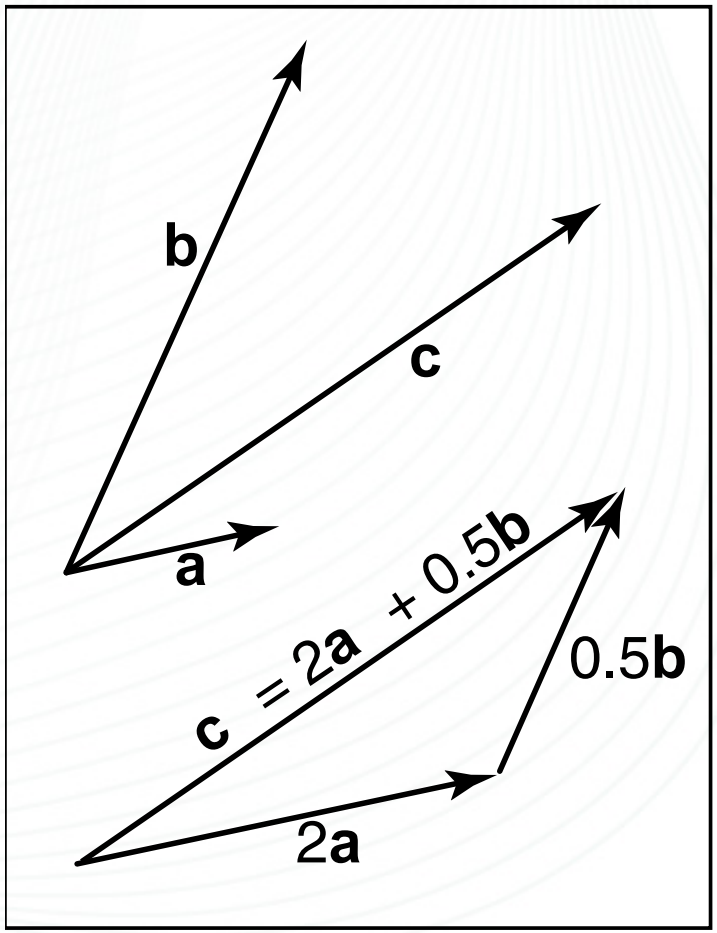
- Um vetor pode ser escrito como uma combinação de dois vetores não nulos que não são paralelos. Se isso ocorrer, os dois vetores mencionados são chamados linearmente independentes.
- Dois vetores linearmente independentes formam uma base 2D e os vetores são referidos como vetores base.

Coordenadas cartesianas de um vetor 2D

- Por exemplo, um vetor c pode ser expressado como a combinação dos vetores a e b.

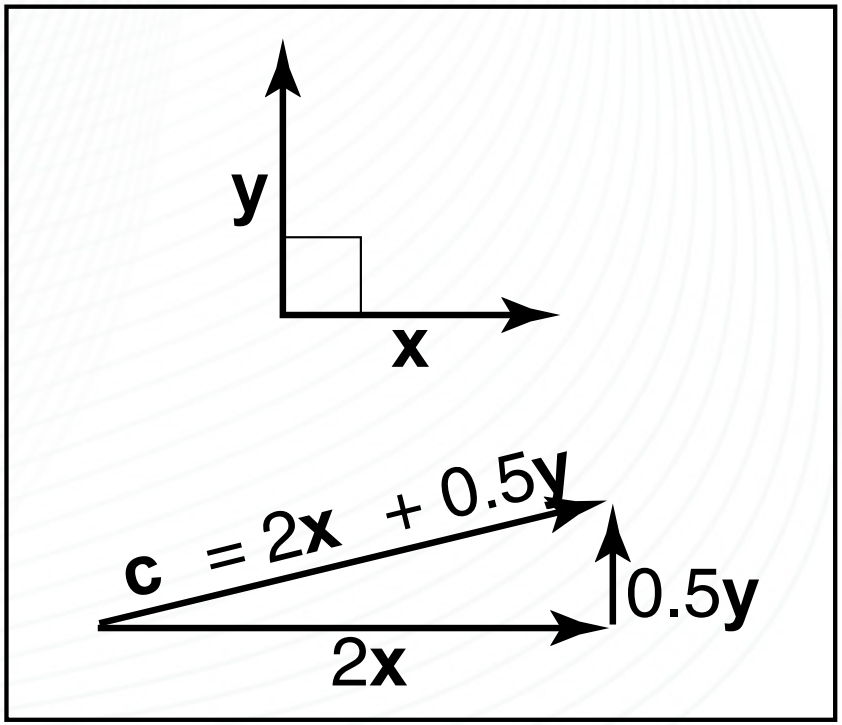
$$\mathbf{c} = a_c\mathbf{a} + b_c\mathbf{b}$$

- Bases podem ser interessantes quando os vetores são ortogonais, ou seja, formam um ângulo reto.



Coordenadas cartesianas de um vetor 2D

- Bases podem ser ainda mais interessantes se os vetores são ortonormais, ou seja, os ângulos são unitários.
- Uma aplicação simples do conceito é o cálculo do comprimento do vetor, onde este assume o valor da hipotenusa.



Vetores: autoria

Os próximos slides foram baseados no material de autoria do

Professor Ravi Ramamoorthi

Multiplicação em vetores

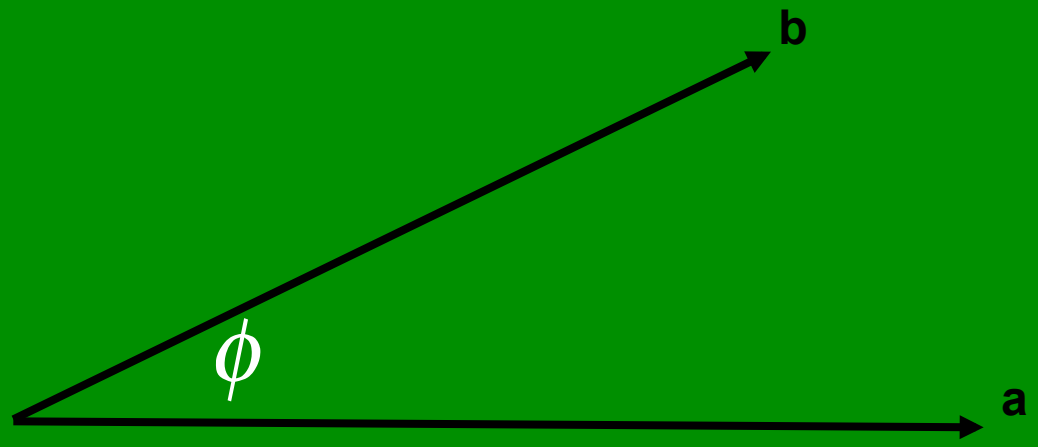
- Produto escalar
- Produto vetorial
- Bases ortonormais e criação de bases

- Nota: alguns livros tratam sobre sistemas de coordenadas da mão direita e da mão esquerda. Nesta disciplina, utilizaremos a regra da mão direita.

Produto escalar

- Produto escalar retorna um número.
- Tem as propriedades de comutatividade, distributividade e associatividade.
- Também, pode ser interpretada como a projeção de um vetor em outro.

Produto escalar



$$a \bullet b = b \bullet a = ?$$

$$a \bullet (b + c) = a \bullet b + a \bullet c$$

$$(ka) \bullet b = a \bullet (kb) = k(a \bullet b)$$

$$a \bullet b = ||a|| \, ||b|| \cos\phi$$

$$\phi = \cos^{-1}\left(\frac{a \bullet b}{||a|| \, ||b||}\right)$$

Produto escalar em componentes cartesianas

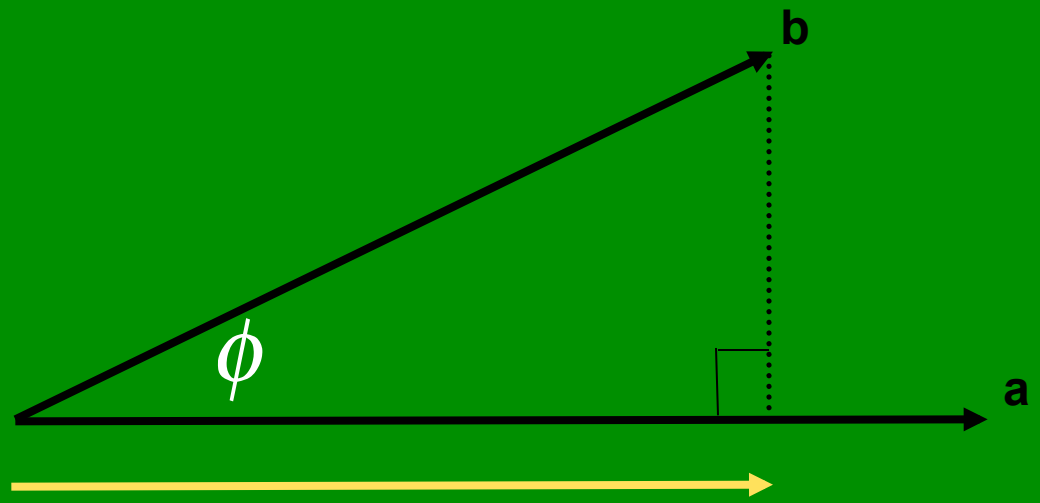
$$a \bullet b = \begin{pmatrix} x_a \\ y_a \end{pmatrix} \bullet \begin{pmatrix} x_b \\ y_b \end{pmatrix} = ?$$

$$a \bullet b = \begin{pmatrix} x_a \\ y_a \end{pmatrix} \bullet \begin{pmatrix} x_b \\ y_b \end{pmatrix} = x_a x_b + y_a y_b$$

Produto escalar: algumas aplicações em CG

- Encontrar o ângulo entre dois vetores (por meio do cosseno do ângulo entre a fonte de luz e uma superfície – para sombra).
- Encontrar a projeção de um vetor em outro (por meio das coordenadas dos pontos em um sistema arbitrário de coordenadas).
- Vantagem: calculado facilmente em componentes cartesianas.

Projeções (de b em a)



$$||b \to a|| = ?$$

$$b \to a \quad = ?$$

$$||b \to a|| = ||b|| \cos\phi = \frac{a \bullet b}{||a||}$$

$$b \to a = ||b \to a|| \frac{a}{||a||} = \frac{a \bullet b}{||a||^2} a$$

Exemplos

u = <1, 4, -3>

v = <-1, 2, 0>

u • v = ?

m = <7, 3, 5>

n = <-8, 4, 2>

m • n = ?

j = <4, 1, 6>

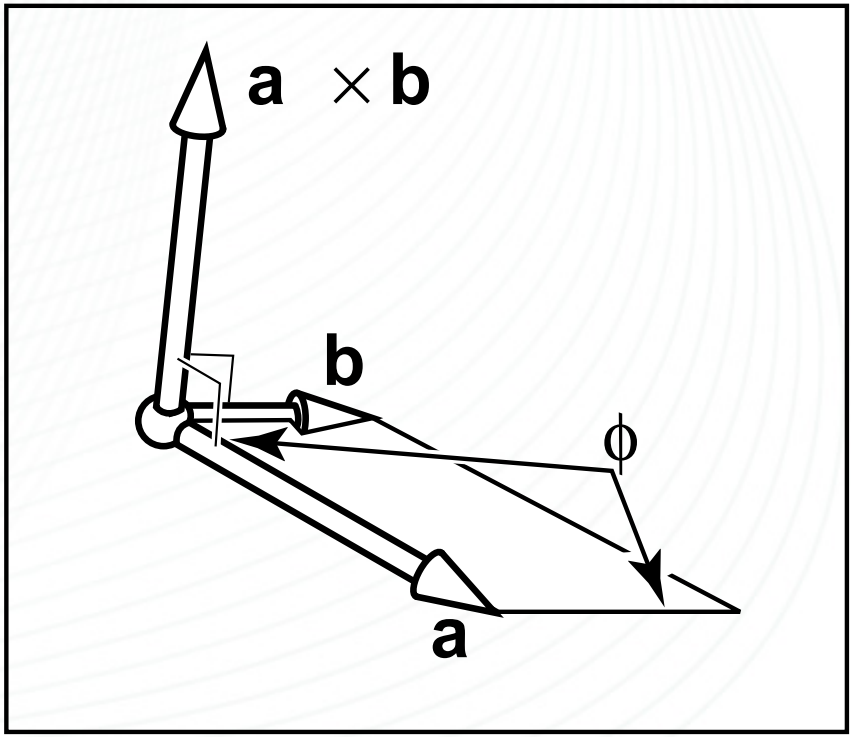
k = <-3, 0, 2>

j • k = ?

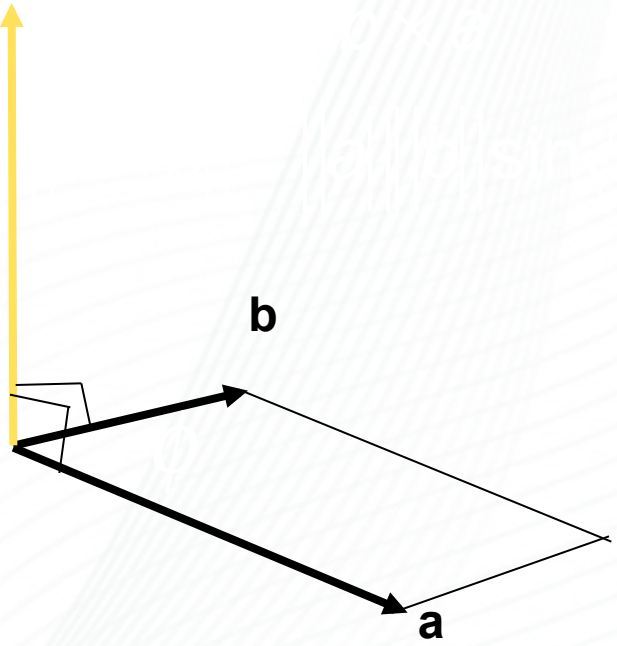
- Se o produto escalar for positivo, o ângulo entre os vetores é agudo (menor do que 90 graus).
- Se o produto escalar for negativo, o ângulo entre os vetores é obtuso (maior do que 90 graus).
- Se o produto escalar for zero, o ângulo entre os vetores é reto (igual a 90 graus)

Produto vetorial

- O produto vetorial retorna um vetor, que é perpendicular aos dois vetores passados como argumentos.

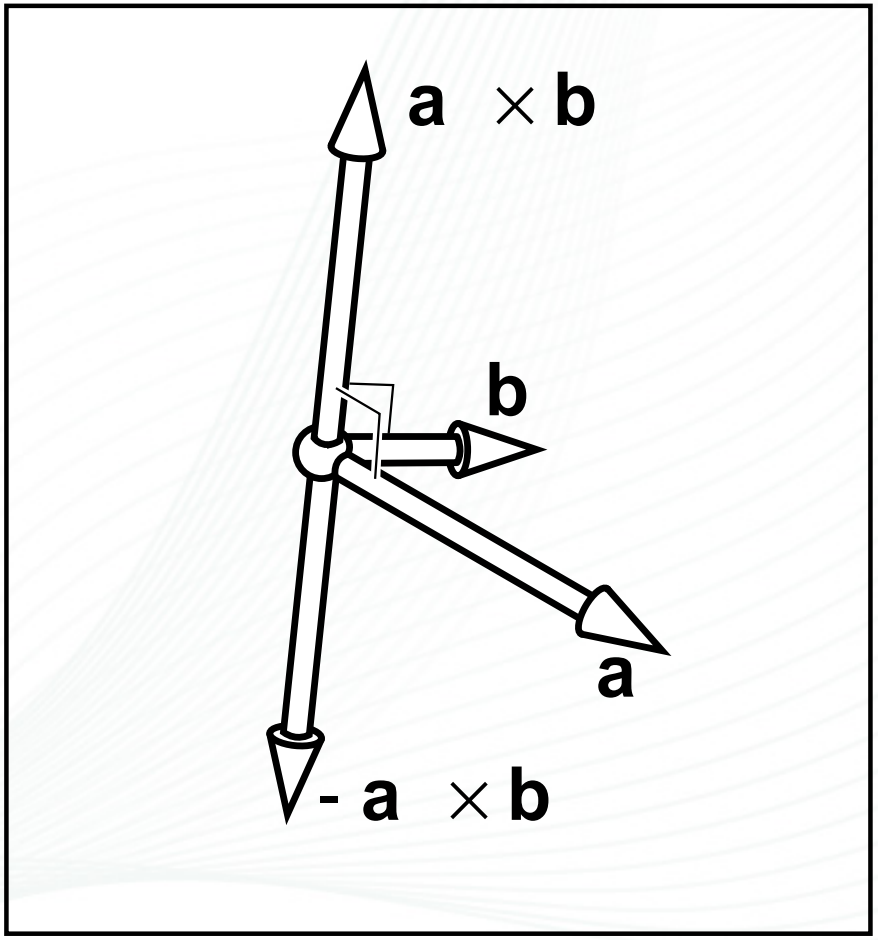


Produto vetorial



- Produto vetorial é ortogonal aos dois vetores
- A direção é determinada pela regra da mão direita (pegar o vetor com a palma da mão)
- Útil na construção de sistemas de coordenadas

Produto vetorial: regra da mão direita



Propriedades do produto vetorial

$$x \times y = +z$$

$$y \times x = -z$$

$$y \times z = +x$$

$$z \times y = -x$$

$$z \times x = +y$$

$$x \times z = -y$$

$$a \times b = -b \times a$$

$$a \times a = 0$$

$$a \times (b + c) = a \times b + a \times c$$

$$a \times (kb) = k(a \times b)$$

Produto escalar: fórmula cartesiana

$$a \times b = \begin{vmatrix} x & y & z \\ x_a & y_a & z_a \\ x_b & y_b & z_b \end{vmatrix} = \begin{pmatrix} y_a z_b - y_b z_a \\ z_a x_b - x_a z_b \\ x_a y_b - y_a x_b \end{pmatrix}$$

Exemplo

u = <1, 2, -2>

v = <3, 0, 1>

$$\begin{bmatrix} i & j & k \\ 1 & 2 & -2 \\ 3 & 0 & 1 \end{bmatrix}$$

u x v = determinante da matriz

u x v = ((2*1) – (0*-2))i + ((-2*3) – (1*1))j + ((1*0) - (3*2))k

u x v = 2i + (-7)j + (-6)k

u x v = <2, -7, -6>

Bases ortonormais

- Base ortonormal: dois vetores u e v formam uma base ortonormal se os vetores formam, entre si, um ângulo reto E ambos têm comprimento igual a um.

$$\|u\| = \|v\| = 1 \quad \text{e} \quad u \bullet v = 0$$

- Em vetores 3D:

$$\|u\| = \|v\| = \|w\| = 1 \qquad u \bullet v = v \bullet w = w \bullet u = 0$$

Bases ortonormais / estruturas de coordenadas

Por que estudar?

- Importante para representar pontos, posições e localizações.

- Frequentemente, muitos conjuntos de sistemas de coordenadas existem, não somente X, Y e Z.
  - Global, local, mundo, modelo, partes do modelo (cabeça, mãos, …)

- Problema crítico são as transformações entre sistemas/bases

Estrutura de coordenadas

- Qualquer conjunto de 3 vetores (em 3D) que:

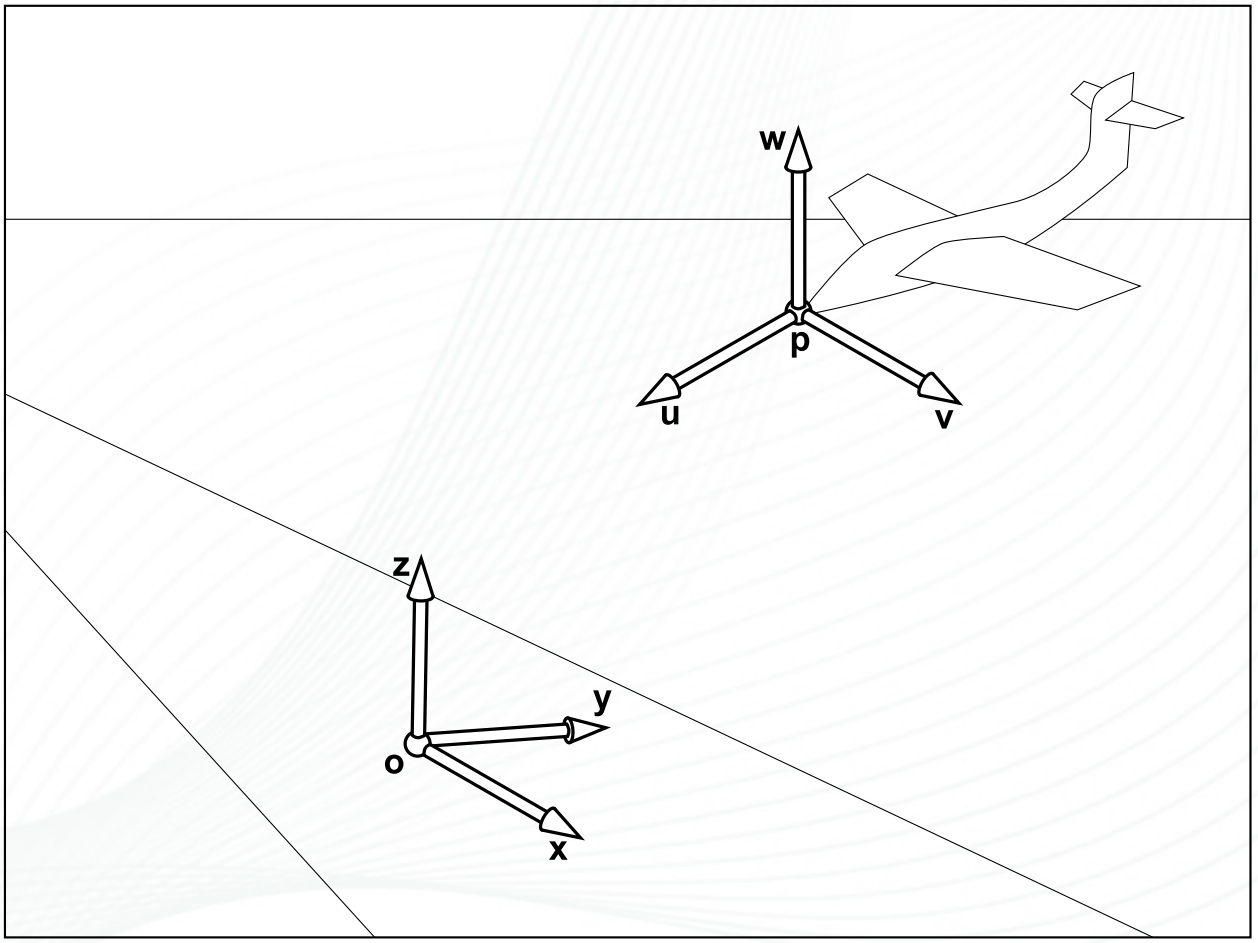
$$\|u\| = \|v\| = \|w\| = 1$$

$$u \bullet v = v \bullet w = u \bullet w = 0$$

$$w = u \times v$$

$$p = (p \bullet u)u + (p \bullet v)v + (p \bullet w)w$$

Estrutura de coordenadas



UFABC

Construindo uma estrutura de coordenadas

- Frequentemente, dado o vetor **a** (por exemplo, a direção de um ponto visível), se quer construir uma base ortonormal).
- É necessário um segundo vetor **b** (por exemplo, a direção da câmera)**.**
- Deve-se construir uma base ortonormal.

Construindo uma estrutura de coordenadas

Nós queremos associar **w** com **a**, e **v** com **b**

- Mas, se **a** e **b** não são ortogonais e nem têm comprimento igual a 1?
- Precisamos encontrar um terceiro vetor **u** que é ortogonal a **b** e **w.**

$$w = \frac{a}{\|a\|} \qquad u = \frac{b \times w}{\|b \times w\|}$$

$$v = w \times u$$

Revisão matemática

- ~~Conjuntos e logaritmos~~
- ~~Equações de segundo grau~~
- ~~Trigonometria~~
- ~~Ponto~~
- ~~Reta~~
- ~~Comprimento de reta~~
- ~~Vetores~~

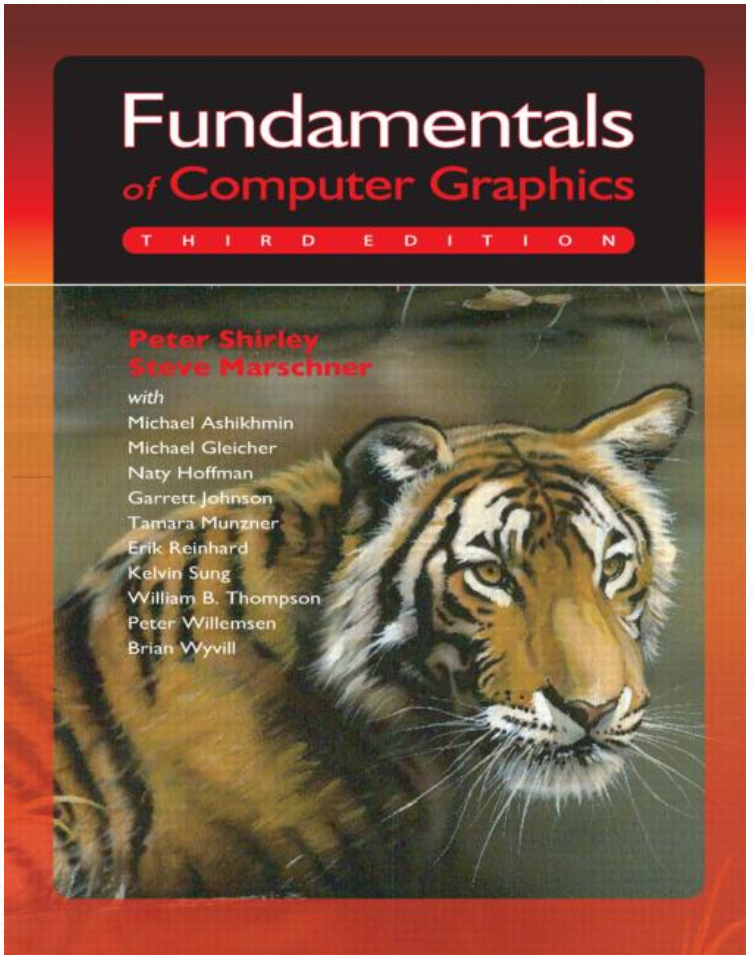
Sumário

- ~~Parte 1~~
  - ~~Continuação da revisão matemática~~
- **Parte 2**
  - **OpenGL**

Aula de hoje

Shirley, Peter, Michael Ashikhmin, and Steve Marschner. Fundamentals of computer graphics. CRC Press, 3rd Edition, 2009.

- **Capítulo 2**



Parte 02

Slides de autoria do
Professor André Balan

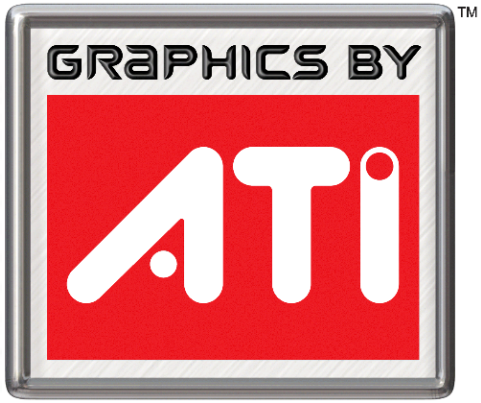
Parte 2

- A evolução tecnológica da computação gráfica ocorreu em duas áreas:
  - Hardware, com o advento e evolução das GPUs
  - Software, com o advento e evolução de APIs gráficas

Hardware - GPU

- O que é uma **GPU**?
- GPU é um acrônimo para "***Graphics Processor Unit***"
- Uma GPU é um processador (*hardware*) dedicado a processar operações para síntese de imagens
- Antigamente, as GPUs eram chamadas de aceleradores gráficos
- Atualmente, praticamente todos os computadores modernos possuem uma GPU, que pode junto com a unidade do processador principal (placa mãe) ou em uma unidade separada (placa de vídeo)

GPUs



- Na década de 90, outras empresas cresceram em entraram na briga pelo mercado de GPUs
- Destacaram-se as, atualmente, renomadas NVidia e ATI
- Principais linhas de GPUs da Nvidia: GeForce e Quadro
- Principais linhas de GPUs da ATI: Radeon e FirePro3D

Software

- Como os programa de computador utilizam uma GPU?

por meio de um **Software: Driver**

- Antes de década de 90, cada fabricante de GPU apresentava sua própria solução de *software* para que os aplicativos comerciais pudessem acessar suas GPUs.
- Resultado: Um determinado aplicativo ficava totalmente amarrado a uma determinada GPU. O Hardware estava em primeiro plano

Software

- A partir da década de 90, este cenário começou a mudar:
  - O Software gráfico passou a ficar em primeiro plano com a popularização das **APIs gráficas**
  - Os desenvolvedores de hardware entenderam que para seguirem no mercado, eles deveriam atender as especificações dessas APIs, e não os seus próprios padrões.

Software

- O que é uma API
  - *Application programming interface*
  - API é a **especificação** de um determinado conjunto de operações/funções computacionais
  - API gráfica é a **especificação** de um conjunto de operações/ funções para síntese de imagens. Exemplo de uma API gráfica bem simples:

```
void linha(int x0, int y0, int xf, int yf);
void circ(int xc, int yc, int rad);
```

OpenGl

- OpenGL significa "*Open Graphics Library*"
- O OpenGL é uma API gráfica livre: "*Open*"
- Começou a ser desenvolvido pela Silicon Graphics (SGI) no início da década de 90, e foi lançado em1992 (ver 1.0)
  - O Direct3D foi lançado em 1995...
- Hoje, todas as GPUs modernas também implementam o OpenGL em hardware, com Drivers para vários SO

OpenGL vs Direct3D

- O Direct3D se popularizou principalmente em meio aos desenvolvedores de jogos:
  - Crysis, Half Life 2, Gears of War, etc...

- OpenGL se popularizou em meio aos criadores de filmes, sistemas CAD, e no meio acadêmico e científico
  - Ambientes de modelagem e animação: 3D Studio, Maya, Adobe After Effects
  - Jogos: Call of Dutty, Max Payne, Half Life 1
  - Visualização científica

OpenGL vs Direct3D

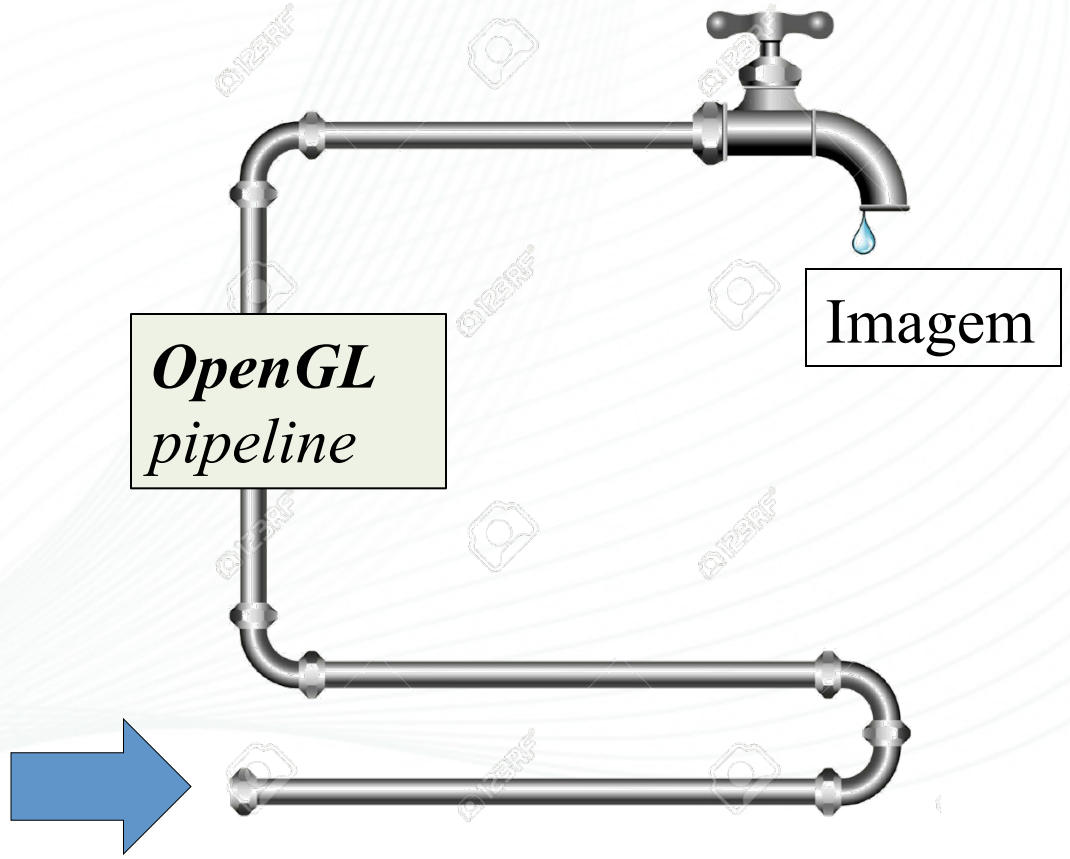
- Neste curso, estudaremos o OpenGL por ser uma API livre e multiplataforma

OpenGL

- OpenGL não é uma linguagem de programação
- OpenGL é uma API gráfica originalmente definida em linguagem C, mas também há definições em diversas outras linguagens.

OpenGL Pipeline

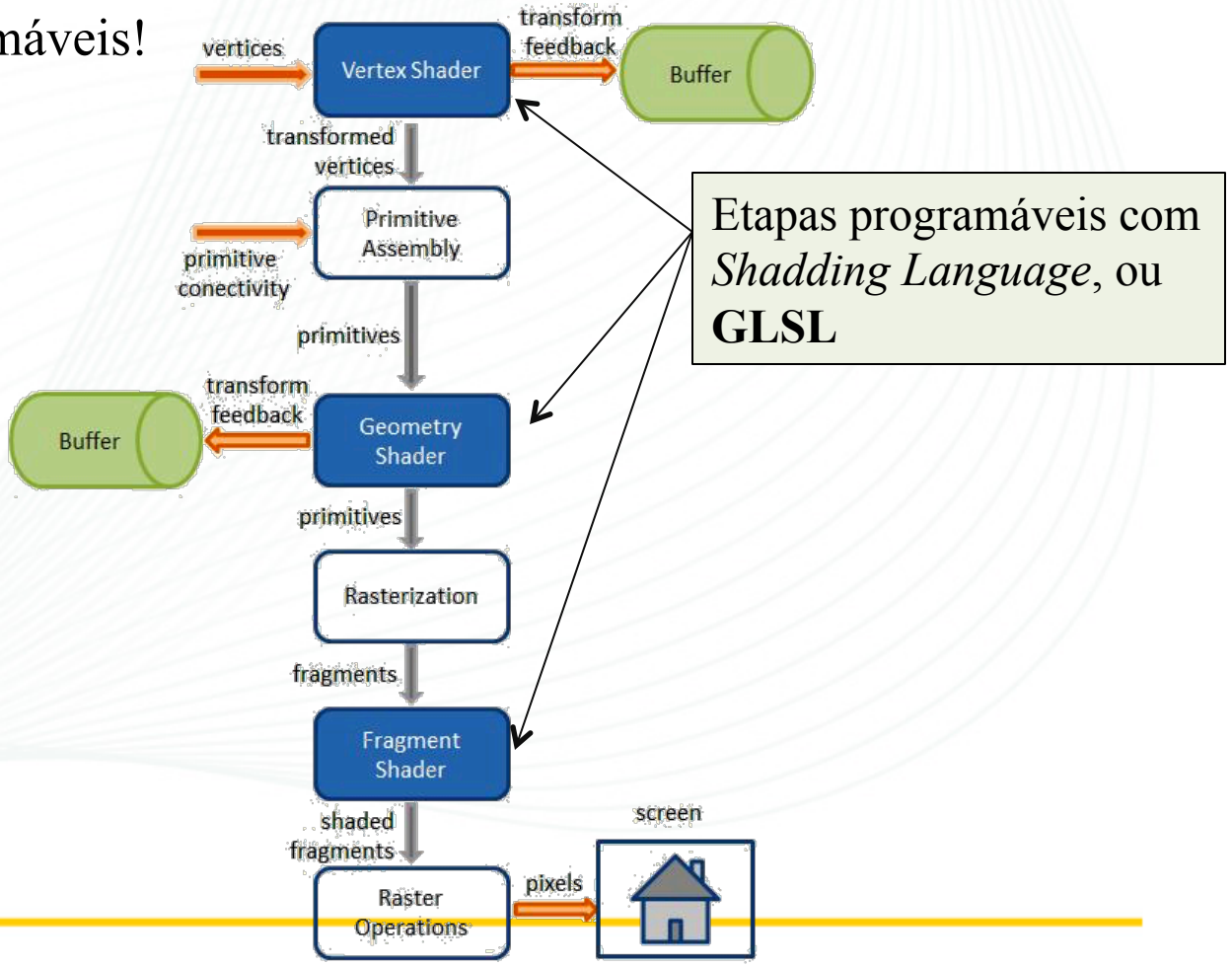
- A estratégia do OpenGL é processar os modelos de primitivas gráficas por meio de várias etapas, como se fosse um processador em formato de tubo!



OpenGL Pipeline

- Com o passar dos anos, esse pipeline foi evoluindo, com épocas de grandes modificações, como em 2004 (versão 2.0), por exemplo, onde suas etapas passaram a ser programáveis!

- Pipeline versão 3.3
  - Bem parecido com a atual versão 4.5



OpenGL

- A API OpenGL especifica mais de 100 funções, que nos permitem, principalmente
  - **Desenhar primitivas** gráficas
    - pontos, segmentos de reta, polígonos,
  - Realizar **transformações geométricas**:
    - translação, rotação e escala nas primitivas
  - Adicionar **iluminação** à cena

OpenGL

- Para **rodar programas que utilizam OpenGL**, precisamos apenas da implementação:
  - No SO Windows, a implementação do OpenGL está na dll **opengl32.dll**
  - No Linux, a implementação OpenGL vem junto com o pacote Mesa3D

OpenGL

- Para **programar** em OpenGL na linguagem C precisamos
  - Da API em si: arquivo **gl.h**
  - Da biblioteca estática: arquivo **opengl32.lib** ou opengl32.a

- O que a opengl32.lib contém?
  - Contém código pre-compilado que acessa a implementação OpenGL presente no computador , ou seja, o arquivo dll.

OpenGL

- O que a API OpenGL não fornece?
  - Operações para:
    - Criação de objetos: cubo, esfera, cilindro, etc...
    - Criação de janelas
    - Tratamento de eventos do teclado, mouse, etc...
    - Realizar animações

OpenGL

- Funções para realizar estas operações são oferecidas por outras APIs relacionadas
  - **GLUT, GLFW, SDL, QT** – criação de janelas, tratamento de eventos do mouse e teclado, criação de objetos poligonais
  - **GLU** – câmera virtual, superfícies curvas, etc...
  - **GLUI** – criação de botões, caixas de texto, etc...
  - **GLEW** – acesso às extensões OpenGL

OpenGL

- Neste curso vamos usar a combinação GLFW + GLU + GLEW

- Vamos criar um projeto em C++11, de 64 bits, utilizando o compilador MingW e a IDE CodeBlocks

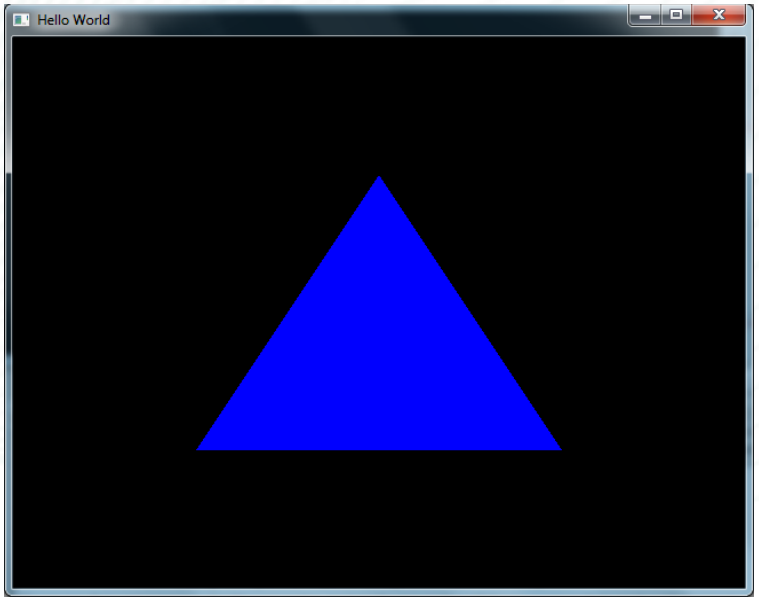
- **Siga os passos do Tutorial 01, 02 e 03**

OpenGL

- **Siga os passos do Tutorial 01, 02 e 03** e volte para essa apresentação

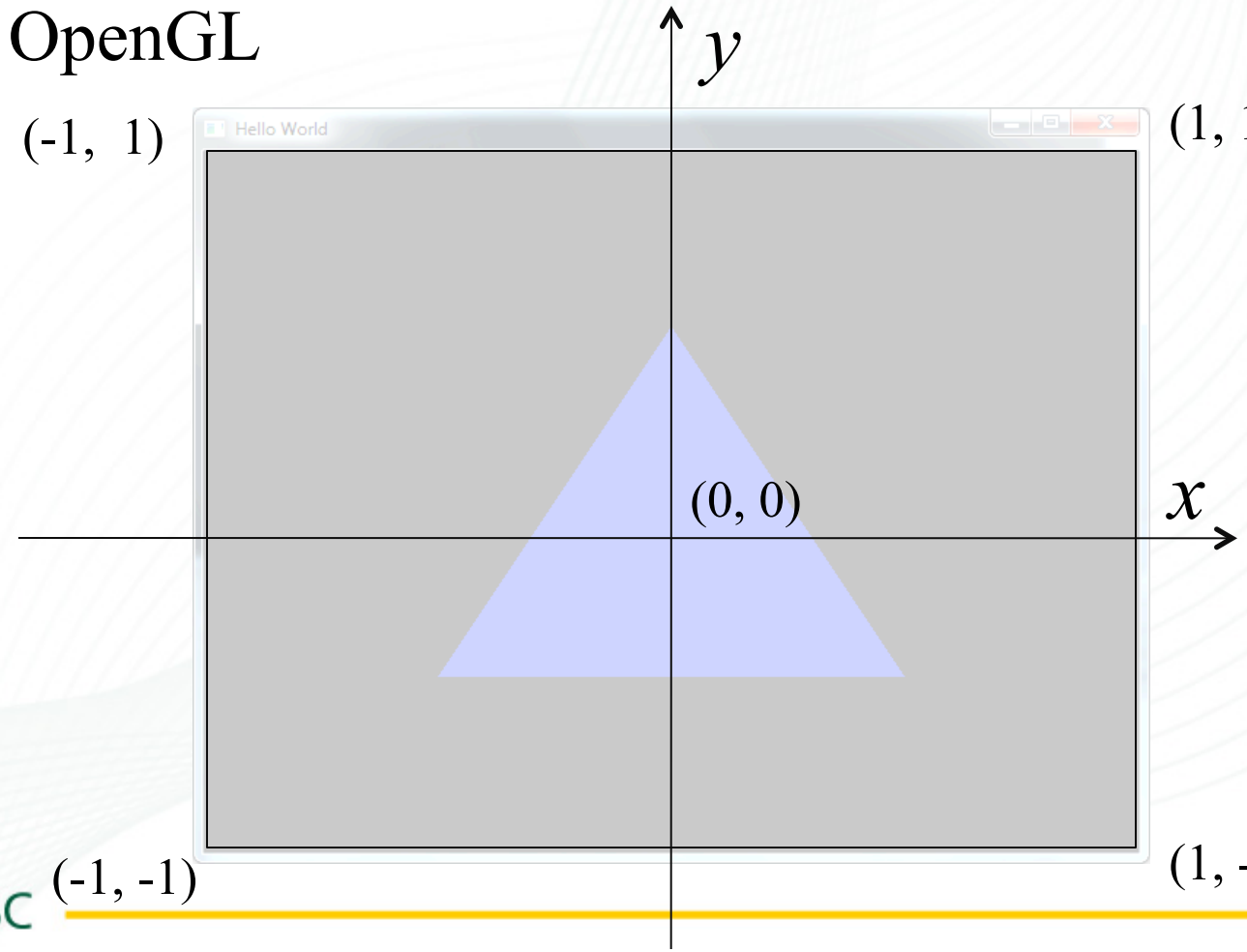
OpenGL

- Momentos depois....
- Até agora não usamos OpenGL!
- Vamos fazer algo simples: Desenhar um triângulo no centro da tela



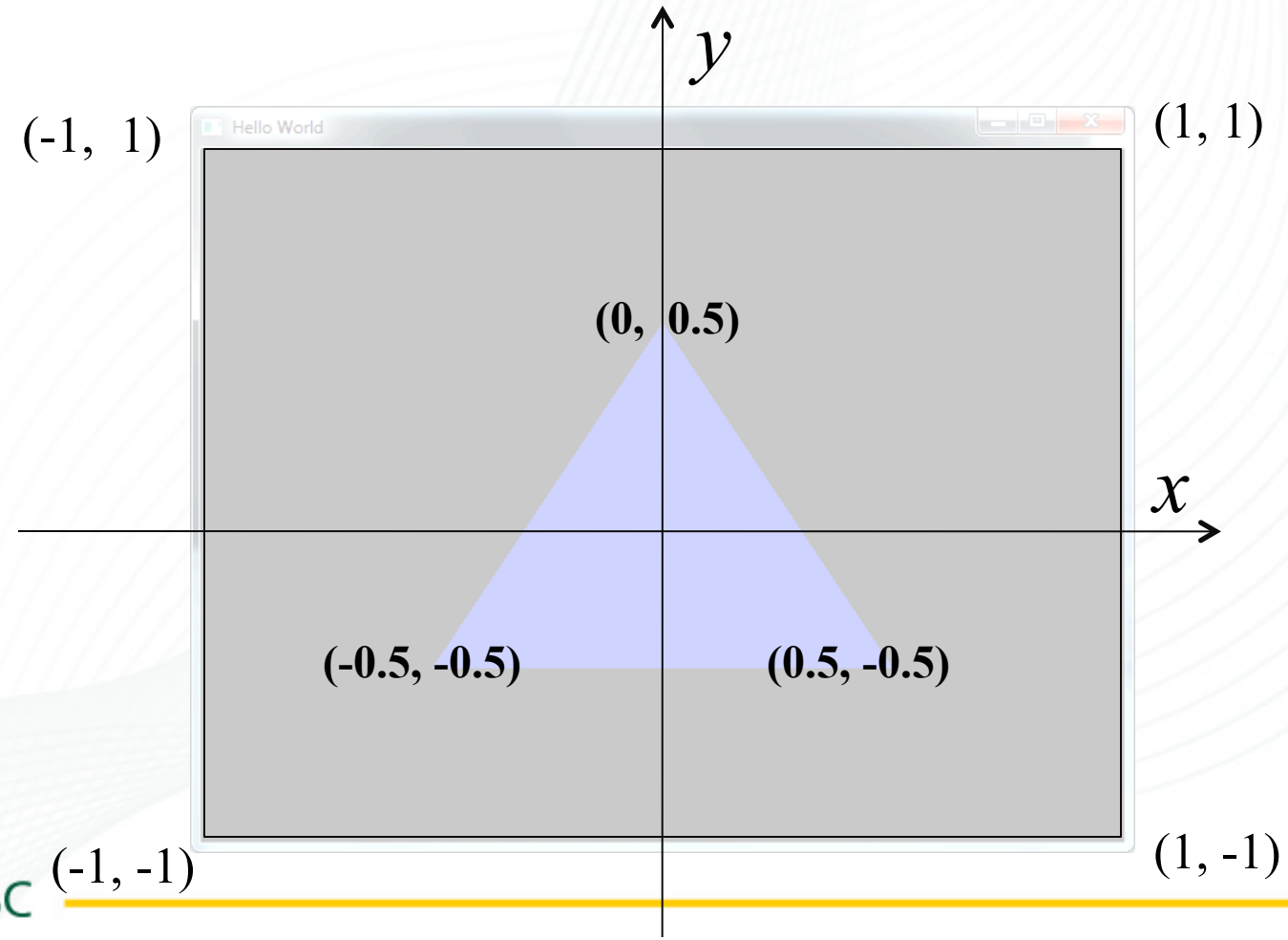
OpenGL

- Inicialmente, observe o sistema de coordenadas padrão do OpenGL



OpenGL

- Precisamos definir os seguintes vértices



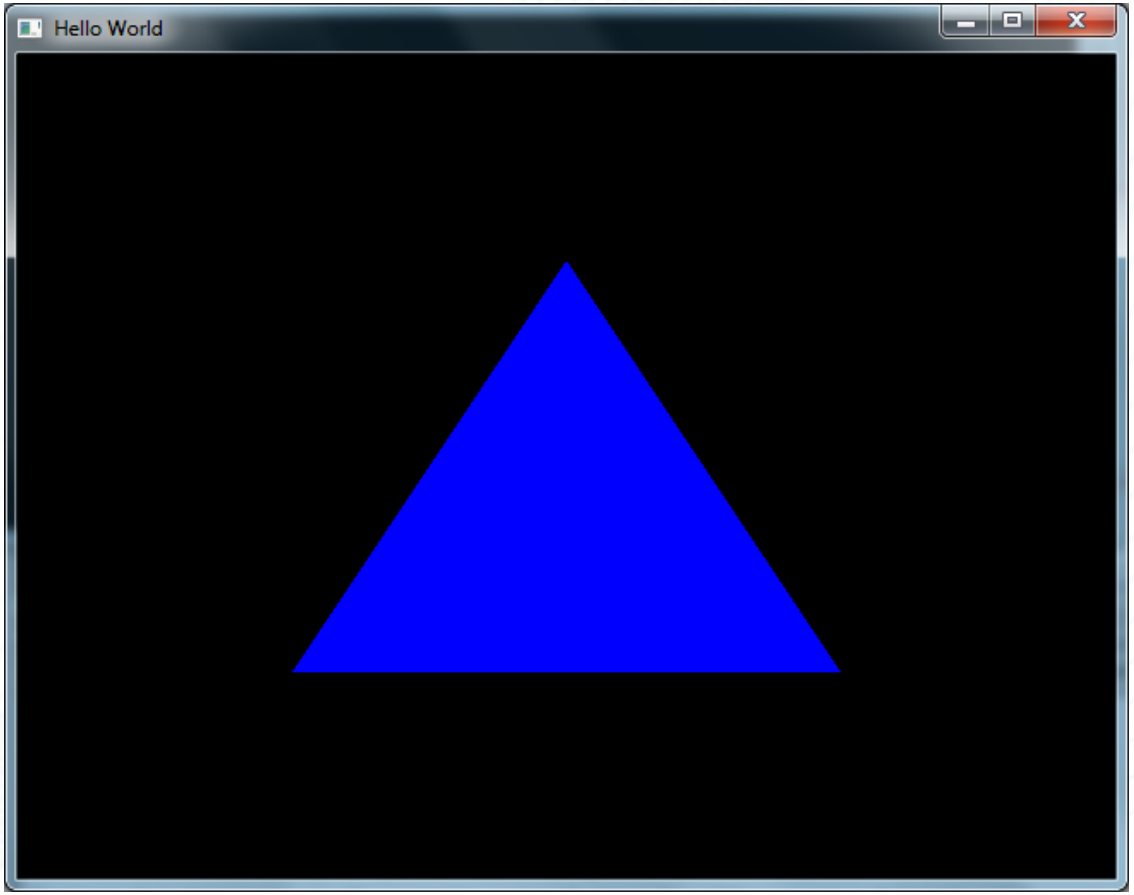
OpenGL

- No loop principal do programa, insira o seguinte código OpenGL (após a chamada da glClear

```
:
// Define a cor atual como sendo azul
glColor3f(0, 0, 1);
// Definimos o estado de montagem de triângulos
glBegin(GL_TRIANGLES);
    glVertex2f(-0.5f, -0.5f);
    glVertex2f(+0.0f, +0.5f);
    glVertex2f(+0.5f, -0.5f);
        /* Estando no estado de criação de triângulos,
       a cada três vértices definidos, o OpenGL
       monta um triângulo  */
glEnd();
:
```

OpenGL

- Resultado



OpenGL

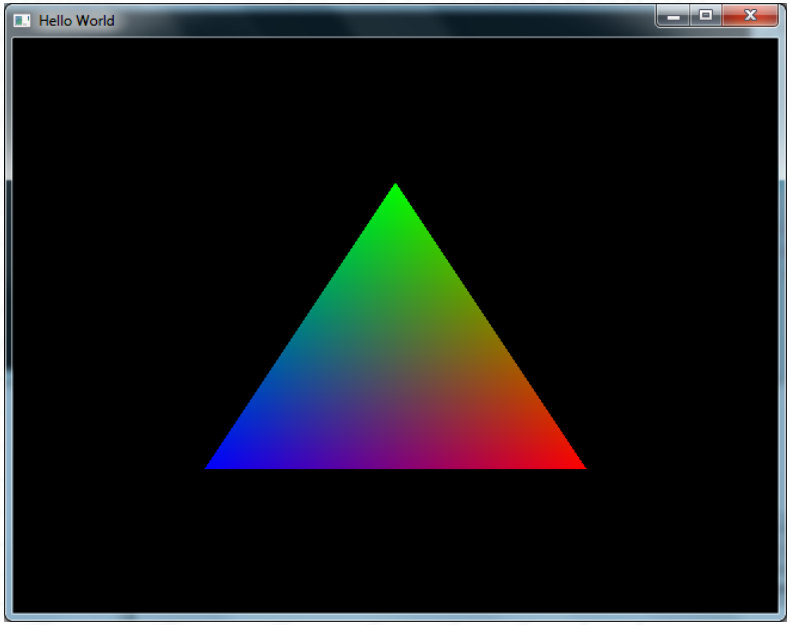
- As funções da API OpenGL foram projetadas de modo a definir uma **máquina de estados**
  - Por exemplo, se quisermos desenhar uma série de linhas na cor azul, precisamos definir a **cor corrente** como azul.
  - Todas as primitivas desenhadas após a definição da cor corrente, serão então desenhadas na cor azul, que determina o **estado atual** da cor
  - Se quisermos desenhar primitivas em outra cor, devemos novamente mudar o estado cor para o qual desejarmos
  - A função que muda a cor atual da máquina é a função glColor.

OpenGL

- Tente definir uma cor diferente antes de definir cada vértice

```
glBegin(GL_TRIANGLES);
  glColor3f(0, 0, 1); // Azul
  glVertex2f(-0.5f, -0.5f);
  glColor3f(0, 1, 0); // Verde
  glVertex2f(+0.0f, +0.5f);
  glColor3f(1, 0, 0); // Vermelho
  glVertex2f(+0.5f, -0.5f);
glEnd();
```

resultado



OpenGL

- O **nome das funções OpenGL** seguem um padrão:
  1. **Prefixo de biblioteca**: gl, glu, glut, etc..
  2. **Comando raiz:** ex: color, vertex, etc..
  3. **Número de parâmetros** a função recebe (opcional)
  4. **Tipo de parâmetros** que a função recebe (opcional)
- Exemplos:
  - glVertex2i - Define um vértice com dois parâmetros inteiros
  - glColor3f - Define a cor corrente com 3 parâmetros reais

OpenGL

- Tipos de dados definidos na API OpenGL

Tipo de dado OpenGL	Representação interna	Tipo equivalente em C	Sufixo
GLbyte	8-bit integer	signed char	b
GLshort	16-bit integer	short	s
GLint	32-bit integer	int ou long	i
GLfloat	32-bit floating-point	float	f
GLdouble	64-bit floating-point	double	d
GLubyte, GLboolean	8-bit unsigned integer	unsigned char	ub
GLushort	16-bit unsigned integer	unsigned short	us
GLuint	32-bit unsigned integer	unsigned int	ui

OpenGL

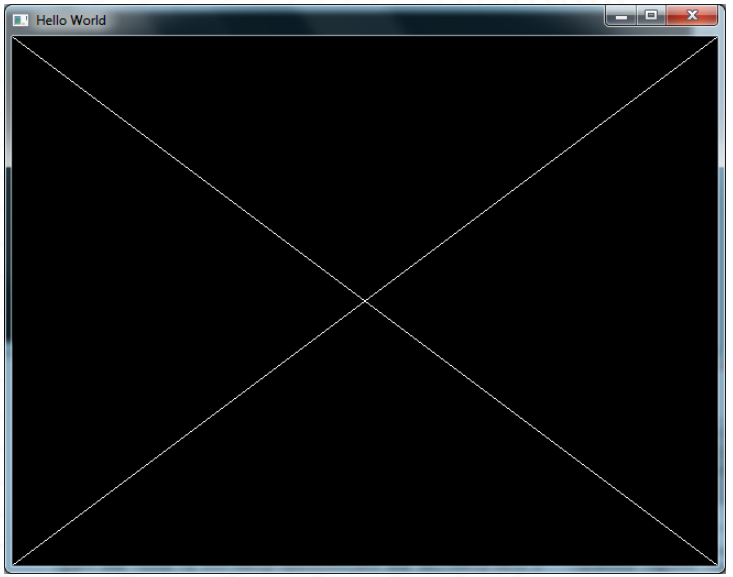
- **Máquina de estados**
  - Outro exemplo: se quisermos desenhar uma série de linhas, devemos colocar a máquina no **estado de desenho de linhas**.
  - Deste modo, sempre que definirmos consecutivamente dois vértices, a máquina irá desenhar um segmento de reta.
  - Mas se estivermos no estado de **desenho de quadriláteros**, por exemplo, quando definirmos dois vértices, a máquina ainda ficará esperando por mais dois vértices, para que ela possa desenhar o quadrilátero.

OpenGL

➢ Máquina de estados

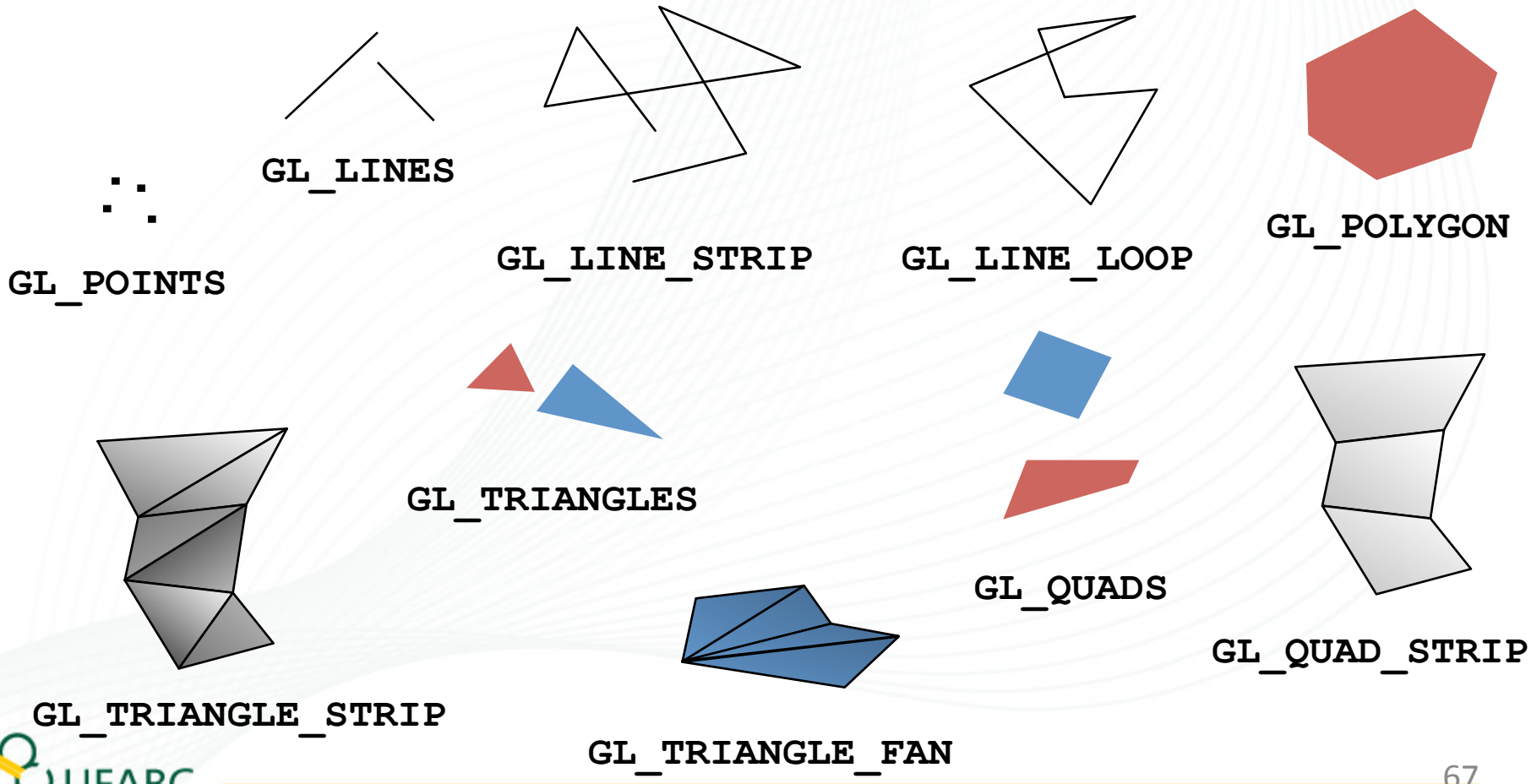
- ➢ Exemplo:

```
:
:
// Define a cor atual como sendo branco
glColor3f(1, 1, 1);
// Definimos o estado de montagem de linhas
glBegin(GL_LINES);
       glVertex2f(-1.0f, +1.0f);
       glVertex2f(+1.0f, -1.0f);
       glVertex2f(-1.0f, -1.0f);
       glVertex2f(+1.0f, +1.0f);
glEnd();
:
:
```



OpenGL

- Definição de estados para todas as primitivas



OpenGL

- Definição de estados para todas as primitivas
  - **GL_LINES**: exibe uma linha a cada dois comandos *glVertex*;
  - **GL_LINE_STRIP**: exibe uma sequência de linhas conectando os pontos definidos por *glVertex*;
  - **GL_LINE_LOOP**: exibe uma sequência de linhas conectando os pontos definidos por *glVertex* e ao final liga o primeiro como último ponto;
  - **GL_TRIANGLES**: exibe um triângulo preenchido a cada três pontos definidos por *glVertex*;
  - **GL_POLYGON**: exibe um polígono convexo preenchido, definido por uma seqüência de chamadas a *glVertex*

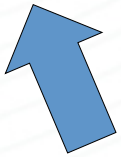
OpenGL

- Definição de estados para todas as primitivas
  - **GL_TRIANGLE_STRIP**: exibe uma sequência de triângulos baseados no trio de vértices v0, v1, v2, depois, v2, v1, v3, depois, v2, v3, v4 e assim por diante;
  - **GL_TRIANGLE_FAN**: exibe uma sequência de triângulos conectados baseados no trio de vértices v0, v1, v2, depois, v0, v2, v3, depois, v0, v3, v4 e assim por diante;
  - **GL_QUADS**: exibe um quadrado preenchido conectando cada quatro pontos definidos por *glVertex*;
  - **GL_QUAD_STRIP**: exibe uma sequência de quadriláteros conectados a cada quatro vértices; primeiro v0, v1, v2, v3, depois, v2, v3, v4, v5, depois, v4, v5, v6, v7, e assim por diante

OpenGL

- **Capturando eventos do teclado com GLFW**
  - Associamos à janela uma função que será invocada automaticamente quando um evento do teclado for gerado. Essas funções chamam-se **callback functions.**
  - A chamada, a seguir, deve estar antes do loop while(!glfwWindowShouldClose())

```
glfwSetKeyCallback(window, key_callback);
```



Ponteiro para a janela do GLFW

Função CallBack
precisa ser implementada exatamente com ***a assinatura esperada***

OpenGL

- **Capturando eventos do teclado com GLFW**
  - Exemplo de implementação da função Callback
    - Implementar a função key_callback antes da função principal do programa.
  - Criar as variáveis xc e yc, que devem ser declaradas como globais do tipo float.

assinatura obrigatória da função callback do teclado:

```
void   (GLFWindow*, int, int, int)
```

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods) {
    if (key == GLFW_KEY_LEFT)  xc = xc - 0.01;
    if (key == GLFW_KEY_RIGHT) xc = xc + 0.01;
    if (key == GLFW_KEY_UP)    yc = yc + 0.01;
    if (key == GLFW_KEY_DOWN)  yc = yc - 0.01;
}
```

OpenGL

- **Capturando eventos do teclado com GLFW**
  - Agora utilize as variáveis globais xc e yc para controlar a posição do triângulo pelas teclas de direção, não esquecendo de limpar a tela antes de redesenhar o triângulo em outra posição.

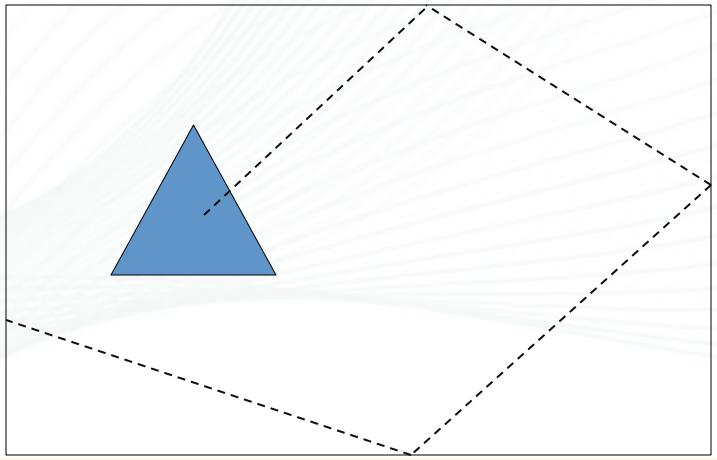
```
glVertex2f(xc-0.5f, yc-0.5f);
glVertex2f(xc+0.0f, yc+0.5f);
glVertex2f(xc+0.5f, yc-0.5f);
```

GL_COLOR_BUFFER_BIT *é um número inteiro que se refere ao buffer de cores do OpenGL (há também o buffer de profundidades, por exemplo)*

Atividade 00

- **Animação**
  - Alterando as variáveis globais xc e yc no laço principal da aplicação, faça como exercício uma animação que:

1. Faça um triângulo ir e voltar aos extremos da tela na direção horizontal
2. Faça um triângulo ir e voltar aos extremos da tela na direção vertical
3. Faça um triângulo quicar nas extremidades da tela conforme abaixo



OpenGL

- **Na verdade...** as funções glBegin, glEnd, glVertex, glColor pertencem ao antigo pipeline do OpenGL, que não possuía etapas programáveis nem transferência eficiente de dados para a memória de GPU. Elas são ultrapassadas e não devem mais ser usadas
- Por quê? Por motivos de desempenho...
  - Imagine: Toda vez que precisamos definir um vértice ou uma cor, é preciso uma chamada de função (alto custo pra CPU) que transfere dados da memória RAM para a GPU. Com milhões de vértices essa aplicação fica pesadíssima!
  - A partir da versão 3.1 surgiu uma forma mais eficiente de transferir e armazenar dados na memória da GPU: **Vertex Buffer Object (VBO).**

OpenGL

- **Referências**
  - Alguns links:
    - http://www.opengl.org
    - http://www.opengl.org/documentation/specs/
    - http://www.inf.pucrs.br/~manssour/OpenGL/Tutorial.html

Fim da Aula 02

André Luiz Brandão