INTRODUÇÃO AO OPENGL

Computação Gráfica e Interfaces

FCT UNIVERSIDADE NOVA DE LISBOA

Sumário

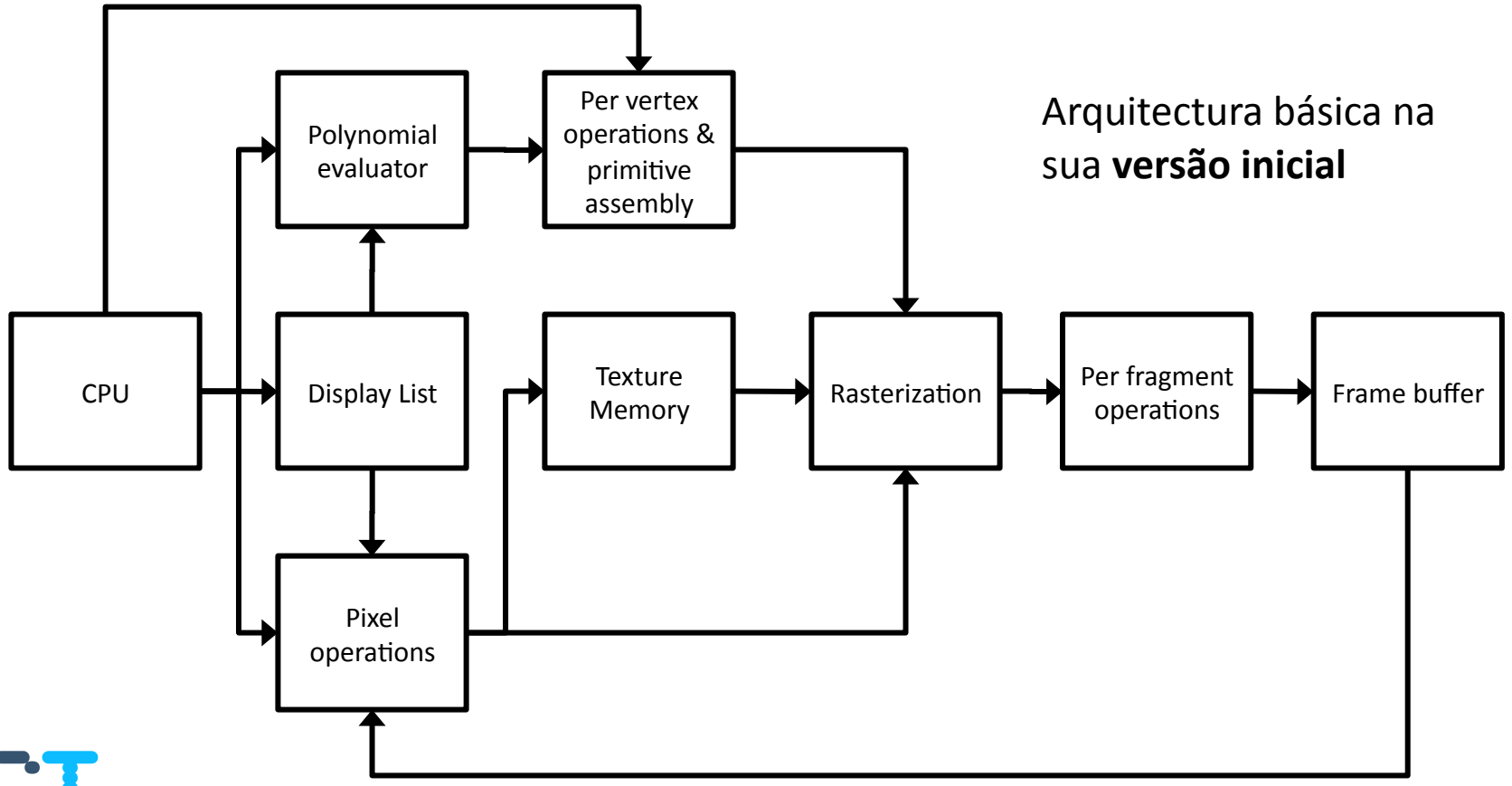
- Arquitectura básica
- Renderização de primitivas
- Transformações
- Animação e profundidade
-
-
-

API de renderização gráfica OpenGL

API standard, de domínio público

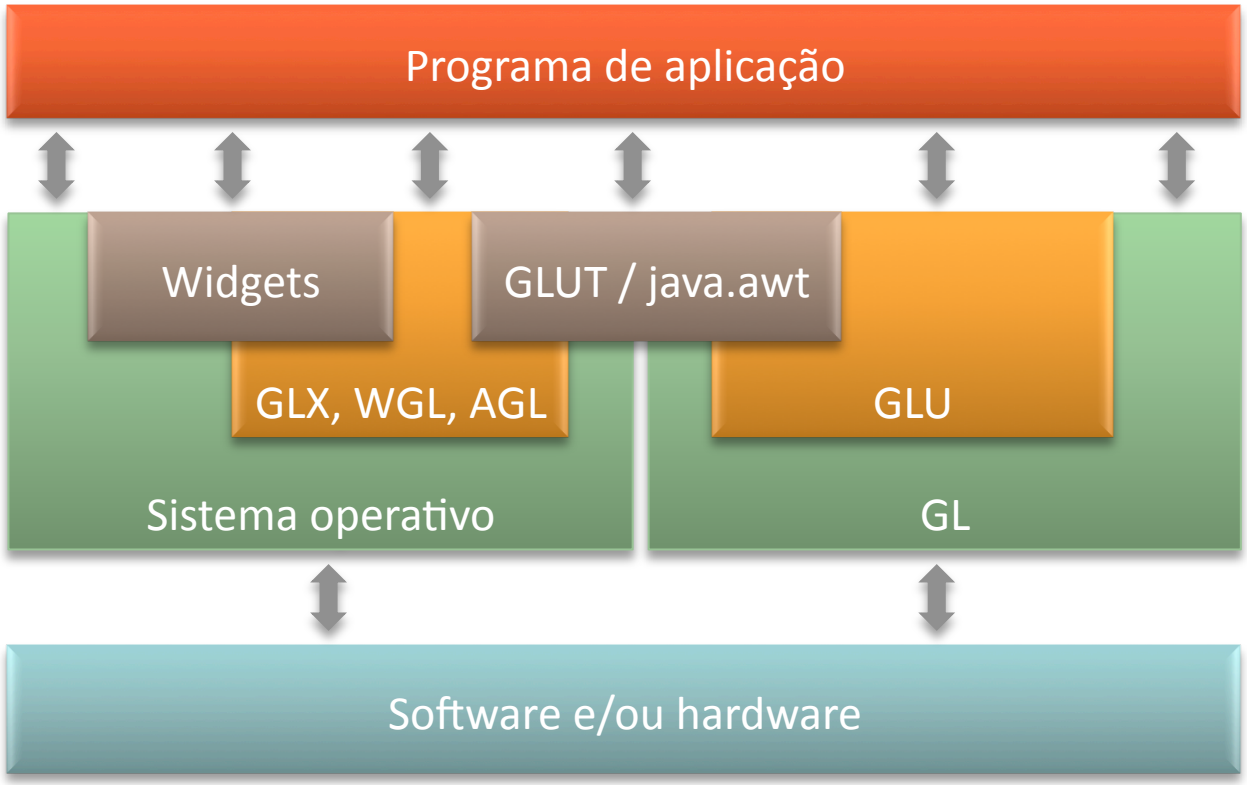
Imagens de cor de boa qualidade compostas por primitivas geométricas e de imagem

Independente do sistema de janelas e do sistema de operação



Arquitectura básica na sua **versão inicial**

Arquitectura de camadas



GLU – OpenGL Utility Library, parte integrante do OpenGL (e.g. NURBS)

GLUT – OpenGL Utility Toolkit, para gestão (simplificada) de janelas e que não é oficialmente parte integrante do OpenGL

GLX, WGL, AGL – bibliotecas adicionais para conversão de uma janela do sistema operativo numa janela OpenGL

Renderização no OpenGL

Primitivas geométricas

- Pontos, linhas e polígonos

Primitivas de imagem

- Imagens e bitmaps
- Separação das *pipelines* de imagens e geometrias, ligadas através de mapeamento de textura

Renderização depende do estado actual

- Cores, materiais, fontes de luz, etc.

Programação em OpenGL

Java.awt / JOGL (java binding for OpenGL)

GLUT/OpenGL

Importação de pacotes JOGL (ou *header files* no caso de GLUT)

- import javax.media.opengl.* (#include <GL/gl.h> …)

Tipos enumerados

- Definição de vários tipos por motivo de compatibilidade: GL.GLfloat, GL.GLint, GL.GLenum, etc.

Estrutura da aplicação

- Configuração e abertura da janela da aplicação
- Inicialização do estado do OpenGL
- Registo de funções para processar eventos de entrada (teclado, rato, etc.)
- Ciclo de processamento de eventos

Em rigor, o OpenGL não foi desenhado numa óptica de programação orientada por objectos – o OpenGL é uma máquina de estados

Java OpenGL (JOGL 1.1.1)

Java OpenGL (JOGL) is a wrapper library that allows OpenGL to be used in the Java programming language.

JOGL allows access to most features available to C programming language programmers, with the notable exception of window-system related calls in GLUT (as Java contains its own windowing systems, AWT and Swing), and some extensions.

OpenGL C API is accessed in JOGL via Java Native Interface (JNI) calls. As such, the underlying system must support OpenGL for JOGL to work. JOGL differs from some other Java OpenGL wrapper libraries in that it merely exposes the procedural OpenGL API via methods on a few classes, rather than attempting to map OpenGL functionality onto the object-oriented programming paradigm. Indeed, the majority of the JOGL code is autogenerated from the OpenGL C header files via a conversion tool named Gluegen, which was programmed specifically to facilitate the creation of JOGL. This design decision has both its advantages and disadvantages. The procedural and state machine nature of OpenGL is inconsistent with the typical method of programming under Java, which is bothersome to many programmers. However, the straightforward mapping of the OpenGL C API to Java methods makes conversion of existing C applications and example code much simpler. The thin layer of abstraction provided by JOGL makes runtime execution quite efficient, but accordingly is more difficult to code compared to higher-level abstraction libraries like Java3D. Because most of the code is autogenerated, changes to OpenGL can be rapidly added to JOGL.

Fonte: JOGL user guide

JOGL 1.1.1 packages:

com.sun.opengl.cg
com.sun.opengl.util
com.sun.opengl.util.j2d
com.sun.opengl.util.texture
com.sun.opengl.util.texture.spi

javax.media.opengl
javax.media.opengl.glu

JOGL numa aplicação Java.awt

1. Criação de uma componente GLCanvas

- *The obvious difference between this code and the typical means of creating AWT components with constructors is that we have to ask a GLCapabilities object for a component that is tuned to the characteristics of the current display, such as its color depth. It might seem odd that the sample uses AWT instead of the more-popular Swing API. While the GLCanvas has a Swing equivalent, GLJPanel, the Swing version does not currently enjoy hardware-accelerated rendering. That means it's slow, which for many would defeat the whole purpose of using JOGL*

Fonte: JOGL user guide

2. Registo do canvas enquanto GLEventListener

- *The GLEventListener is primarily used to call back to our component when it needs to repaint itself or deal with various changes. The interface defines four methods:*
  - *init (GLDrawable drawable): called when OpenGL is initialized, and thus useful for any one-time-only setup work*
  - *display (GLDrawable drawable): a request for the component to draw itself*
  - *reshape (GLDrawable drawable, int i, int x, int width, int height): signals that the component's location or size has been changed*
  - *displayChanged (GLDrawable drawable, boolean modeChanged, boolean deviceChanged): used to signal that the display mode or device has changed (ex: color mode)*

Classe GL

- *The docs define it as "the basic interface to OpenGL." Its design is almost like a straightforward dump of the gl.h header file. In a sense, it's better not to think of it as an object as all, but rather as handle for making method calls. If you're porting from native OpenGL code, then you would expect functions that start with a gl or constants that start with a GL to be accessed via this GL instance. So all we've had to do to port to JOGL is to tack gl. on each of these calls. When a call to glu.h is necessary, we'll get the GLU object from the GLDrawable and make glu.-type calls*

PS. Notice that the GL object is retrieved from the GLDrawable that is passed into the methods . All of the GLEventListener methods provide this object, and one should always get a fresh reference to the GL object from the GLDrawable on each callback rather than caching in a field and reusing it, due to threading issues within AWT and the danger of making OpenGL calls from the wrong thread

The reshape() method sets or re-sets its size and "viewport," which represents what part of the component is being drawn into; in this simple case, this is always the entire component. We also have to make some calls to indicate that we're working in a simplistic 2D environment with no rotations, translations, or other matrix-based transformations

The display() method, a callback that tells the component to perform its drawing. This will be called after init() and reshape() at startup, and again after any GUI events that could require a repaint, such as dragging the component's window around, placing another window over the component, etc. The display() method could also be called by JOGL's Animator class, which exists to call display() repeatedly from a loop. This has obvious use for games, media, and other applications - they perform animation by slightly changing the component on each successive call to display().

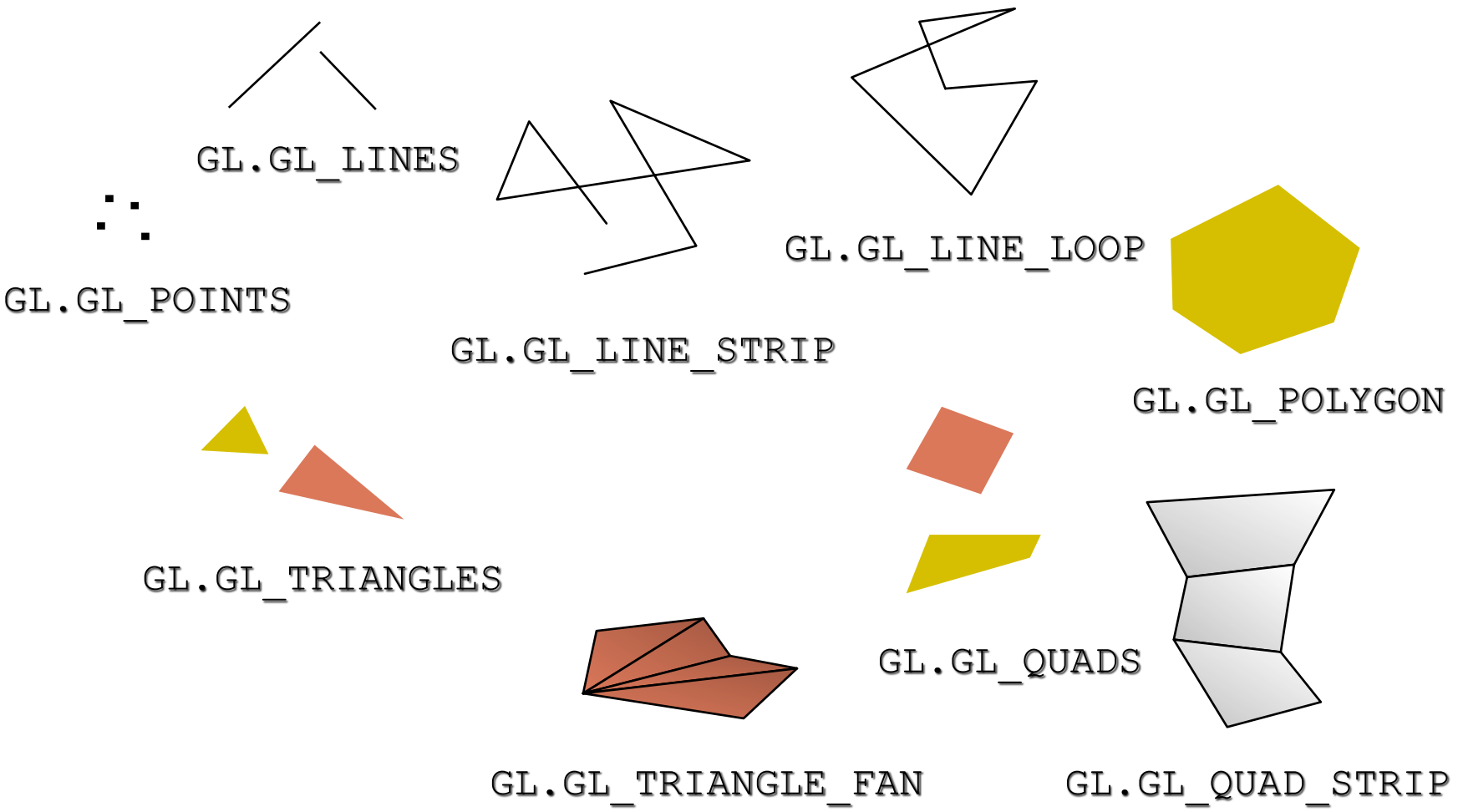
```
...
GL gl = drawable.getGL();
GLU glu = drawable.getGLU();
gl.glViewport( 0, 0, width, height );
gl.glMatrixMode( GL.GL_PROJECTION );
gl.glLoadIdentity();
glu.gluOrtho2D( 0.0, 450.0, 0.0, 375.0);
```

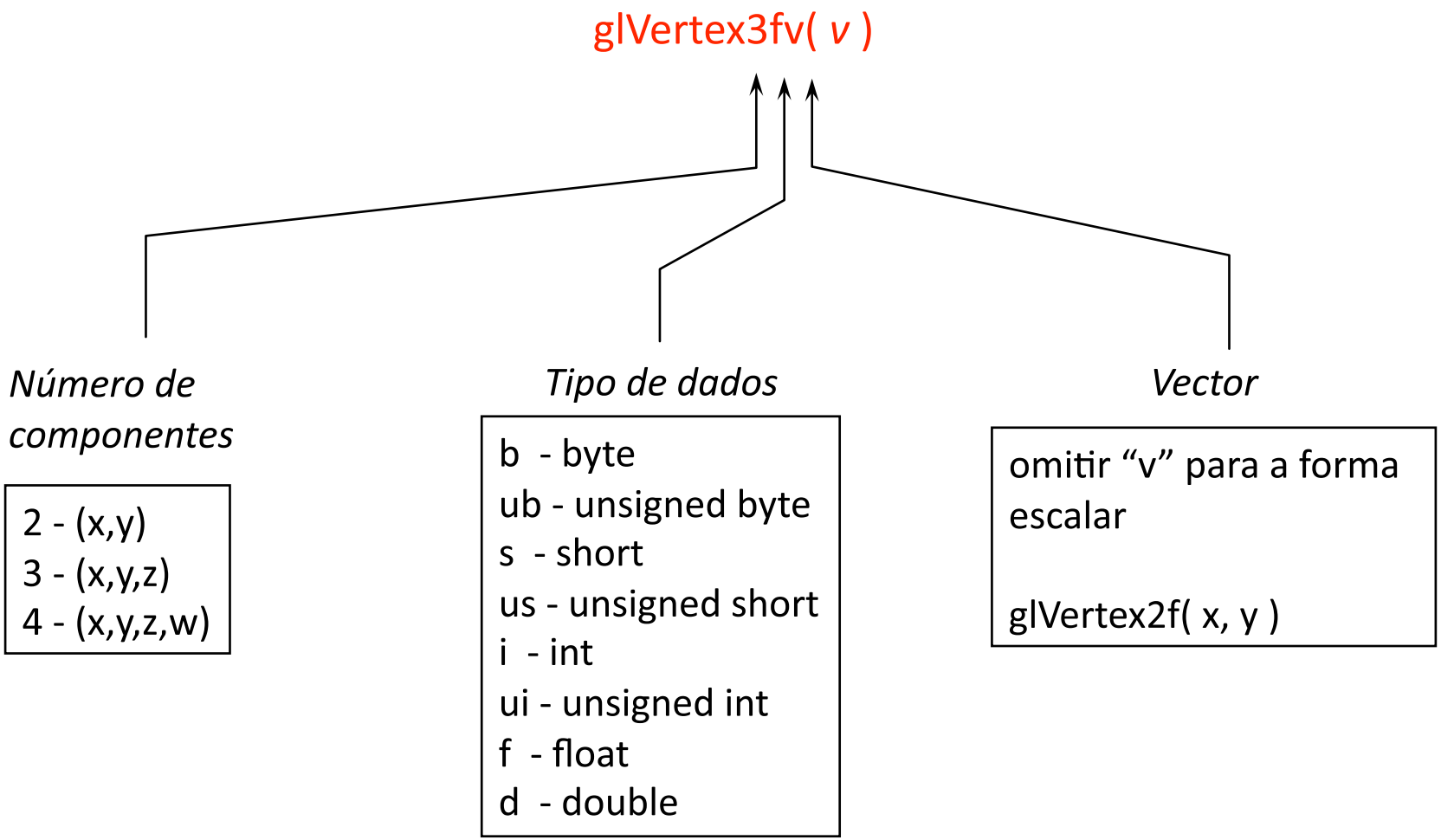
Sumário

- Arquitectura básica
- Renderização de primitivas
- Transformações
- Animação e profundidade

Primitivas geométricas

Primitivas são especificadas através de vértices, em coordenadas homogéneas





Obs.: Por exemplo, o valor de z é automaticamente colocado a 0 no caso de glVertex2f()

As primitivas são especificadas usando o par

```
gl.glBegin( primType );
    ...
gl.glEnd();
```

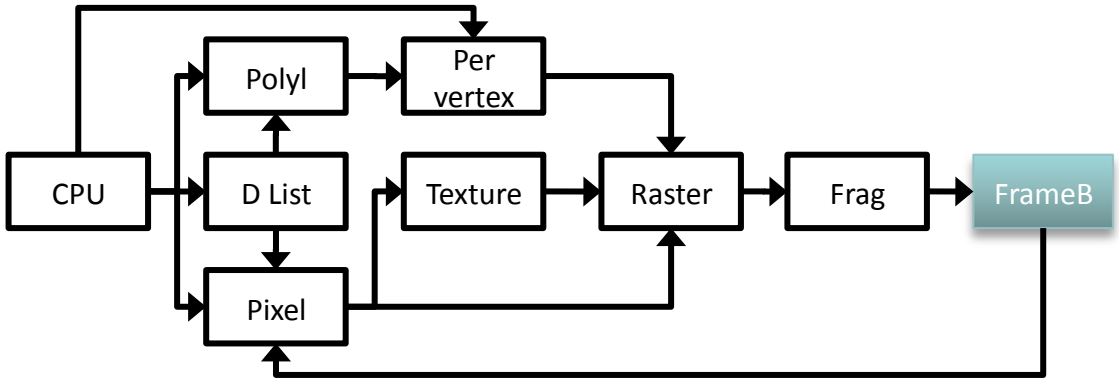
em que *primType* determina o modo de ligação dos vértices

Com JOGL:

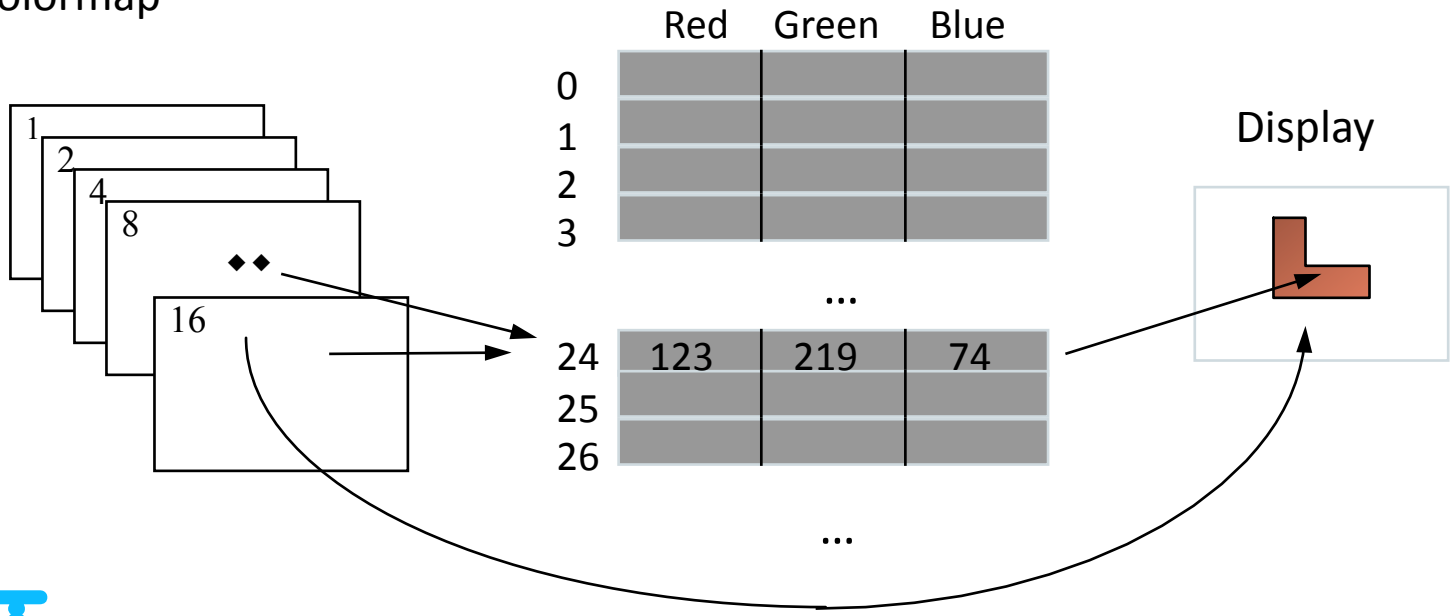
```
gl.glColor3d(0.0, 0.0, 0.0);
gl.glBegin(GL.GL_LINES);
   gl.glVertex2d(120.0,-220.0);  gl.glVertex2d(120.0,-140.0);
   gl.glVertex2d(280.0,-220.0);  gl.glVertex2d(280.0,-140.0);
gl.glEnd();
```

Modos de cor

1. Colormap, através de um índice de uma tabela de cores
2. TrueColor, através da indicação directa da cor

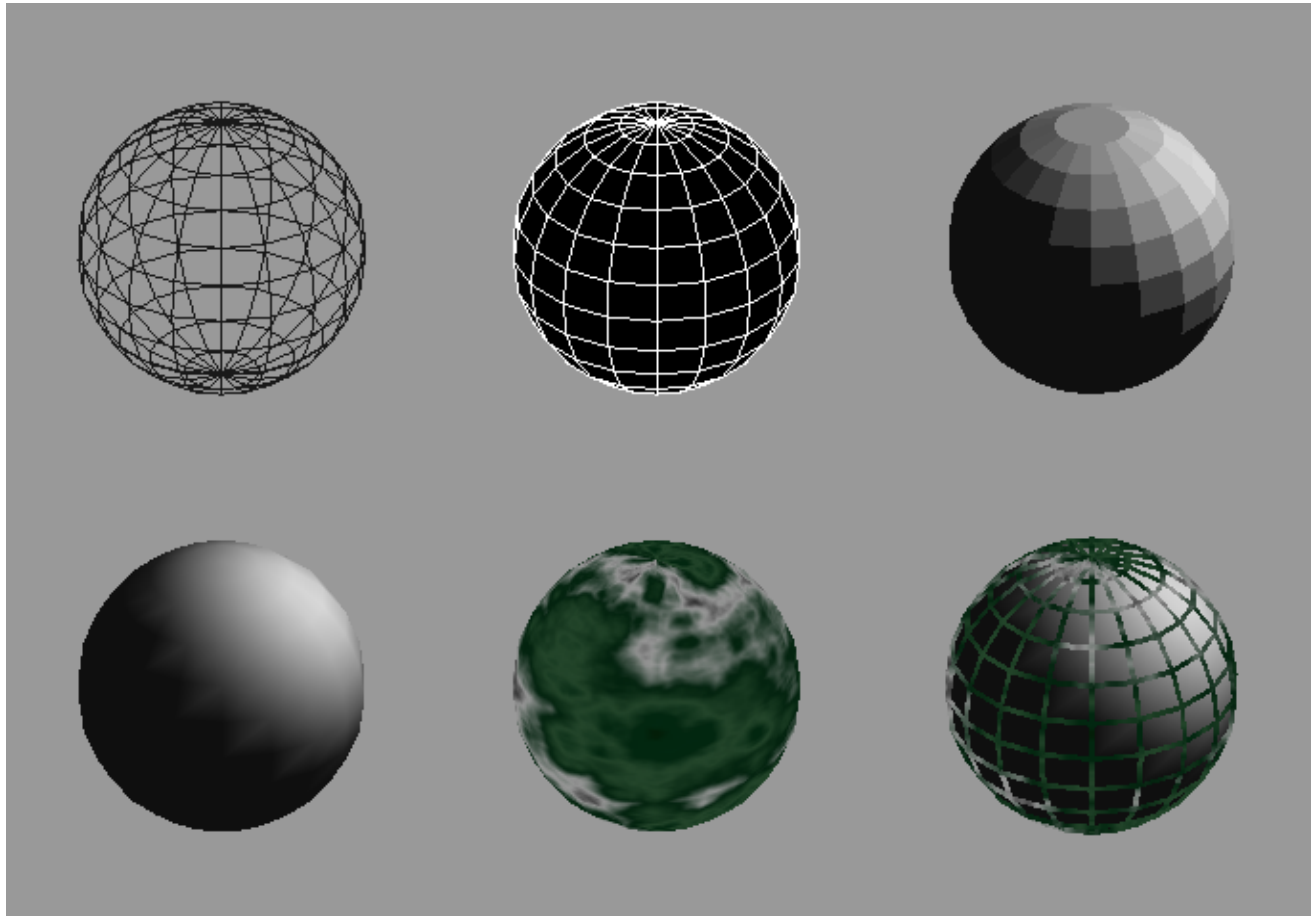


Colormap



Aparência da renderização

Inúmeras formas de renderização: desde malha de linhas até superfícies com iluminação e mapeamento de texturas



Máquina de estados

Os atributos de renderização estão encapsulados no estado do OpenGL

- Estilos de renderização
- Sombreamento
- Iluminação
- Mapeamento de textura

A aparência é controlada pelo estado corrente

A manipulação dos atributos dos vértices é a forma mais usual de afectar o estado

- gl.glColor*() / gl.glIndex*(), gl.gl Normal*(), gl.glTexCoord*()

Definição de estado

- Ex: gl.glPointSize(size), gl.glLineStipple(repeat, pattern), gl.glShadeModel(GL_SMOOTH);

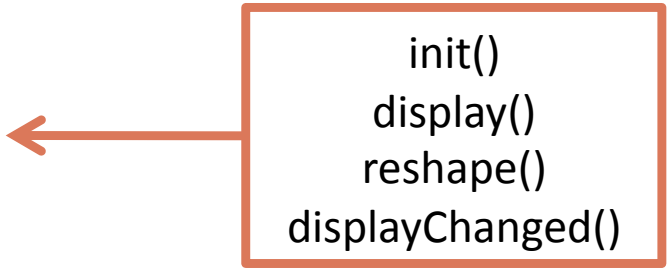
Activação de características

- gl.glEnable(GL_LIGHTING), gl.glDisable(GL_TEXTURE_2D) ;

Exemplo de um programa de desenho de primitivas com JOGL

```
import java.awt.Frame; import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent; import javax.media.opengl.GLEventListener;
import javax.media.opengl.GL; import javax.media.opengl.GLAutoDrawable;
import javax.media.opengl.GLCanvas; import javax.media.opengl.glu.GLU;

public class CasaJOGL implements GLEventListener {

   public void init(GLAutoDrawable gLDrawable) {
     GL gl = gLDrawable.getGL();
     gl.glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
   }
   public void display(GLAutoDrawable gLDrawable) {
     GL gl = gLDrawable.getGL();
     gl.glClear(GL.GL_COLOR_BUFFER_BIT);
     gl.glMatrixMode(GL.GL_MODELVIEW);
     gl.glLoadIdentity();

     gl.glColor3d(0.0, 0.0, 0.0);
     gl.glBegin(GL.GL_LINES);
     gl.glVertex2d(120.0,-220.0);  gl.glVertex2d(120.0,-140.0);
     gl.glVertex2d(280.0,-220.0);  gl.glVertex2d(280.0,-140.0);
     gl.glEnd();
   // …. Outros elementos gráficos
}
```

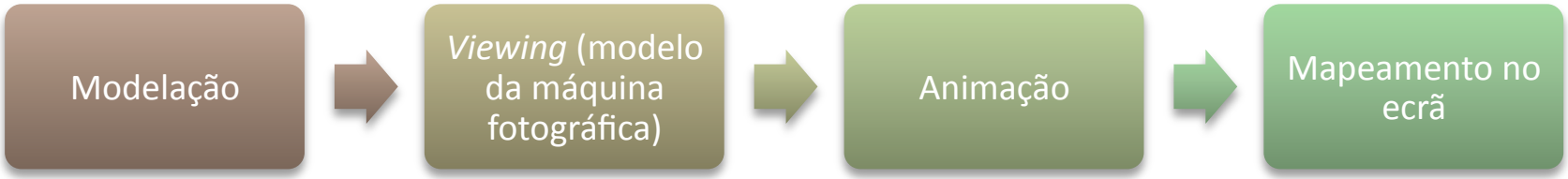
```
public void reshape(GLAutoDrawable gLDrawable, int x, int y, int width, int height) {
      GL gl = gLDrawable.getGL();
       GLU glu = new GLU();
       gl.glViewport(0, 0, width, height);
      gl.glMatrixMode(GL.GL_PROJECTION);
      gl.glLoadIdentity();
      glu.gluOrtho2D(0.0, 400.0, -280.0, 0.0);
   }
   public void displayChanged(GLAutoDrawable gLDrawable, boolean modeChanged,
                  boolean deviceChanged) {}

   public static void main(String[] args) {
      Frame frame = new Frame("Casa JOGL");
      GLCanvas canvas = new GLCanvas();
       canvas.addGLEventListener(new CasaJOGL());
       canvas.setSize(400, 280);
       frame.add(canvas);
      frame.pack();  // subcomponents of window to be laid out at their preferred size
      frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                  System.exit(0);
            }
        } );
      frame.setVisible(true);
   }
}
```



Sumário

- Arquitectura básica
- Renderização de primitivas
- **Transformações**
- Animação e profundidade

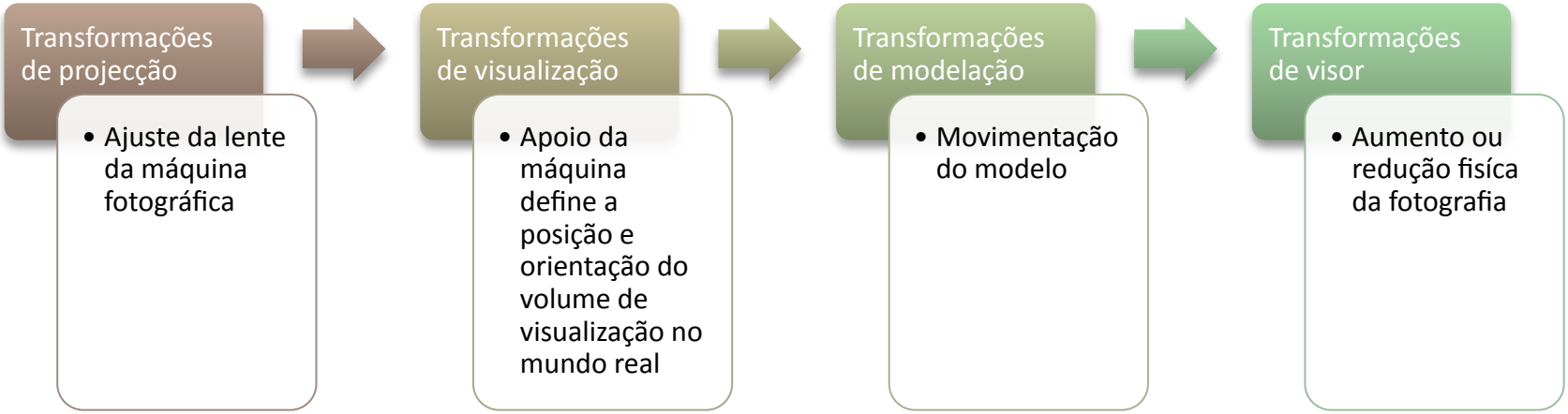
Sequência de transformações



Criação da imagem

- Especificação da geometria (*world coordinates*)
- Especificação da máquina fotográfica (*camera coordinates*)
- Projecção (*window coordinates*)
- Mapeamento no visor (*screen coordinates*)
- Cada um dos passos anteriores recorre a transformações, as quais são equivalentes a alterações de sistema coordenado (*frames*)

Outra perspectiva, em termos de transformações:



Transformações

As transformações devem preservar o tipo de geometria utilizado e.g. linhas, polígonos e quádricas

Transformações afim (preservação de linha)

- Rotação, translação, mudança de escala
- Projecção
- Concatenação ou composição

Vértices e matrizes

- Cada vértice é um vector coluna, normalmente com a coordenada homogénea w a 1
- Todas as operações afim são multiplicações de matrizes, na forma pós-multiplicação
- As direcções (rectas) podem ser representadas com w=0
- Um vértice é transformado por matrizes de dimensão 4x4
- As matrizes são armazenadas em OpenGL coluna a coluna

$$\vec{v} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \qquad \mathbf{M}\,\vec{v}$$

$$\mathbf{M} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

Estilos de especificação de transformações

- Especificação de matrizes (glLoadMatrix, glMultMatrix)
- Especificação de operação (glRotate, glOrtho)

Matrizes

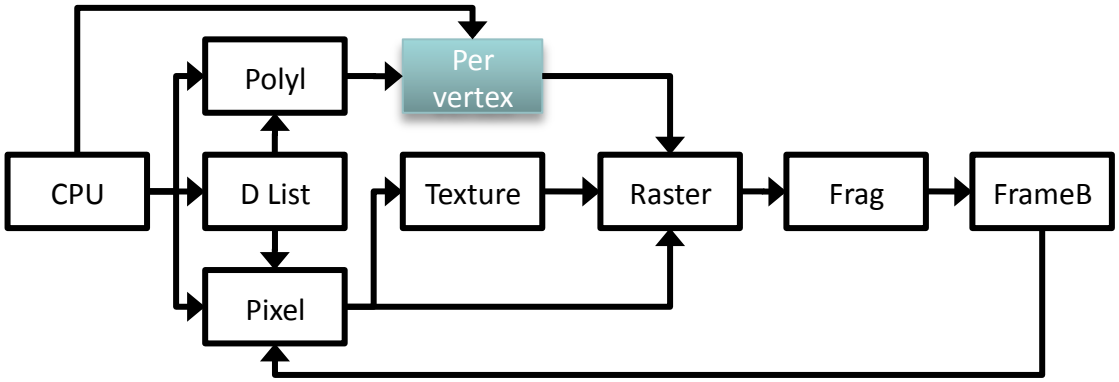
As matrizes devem ser especificadas antes de renderização, visualização, localização e orientação

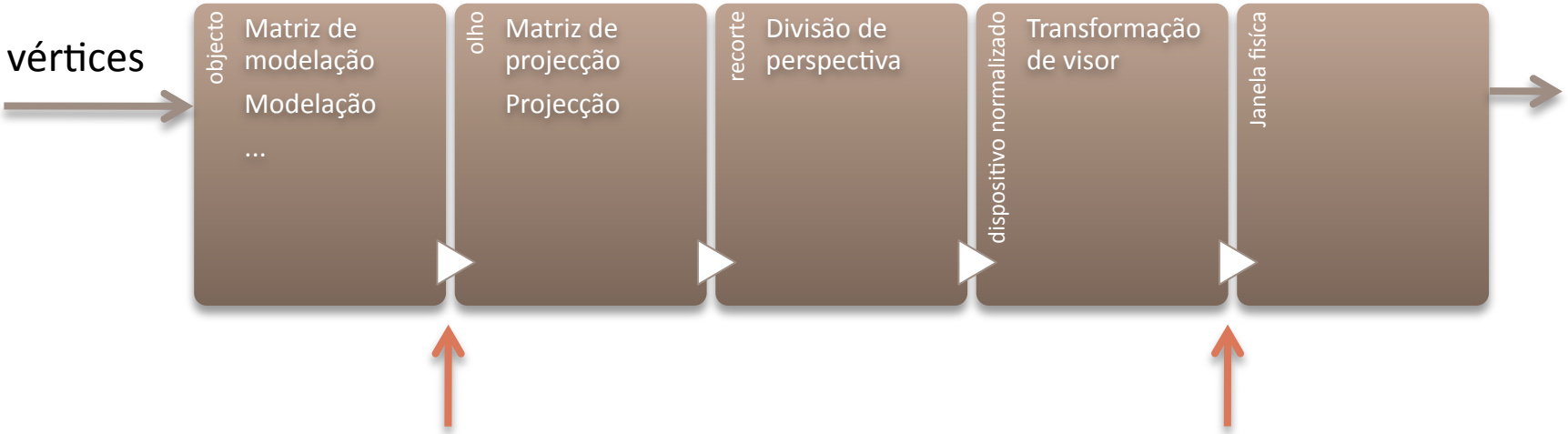
- Posicionamento da máquina fotográfica
- Geometria 3D

As matrizes são sucessivamente manipuladas, incluindo as estruturas de pilha, já que estas são parte integrante do estado do OpenGL

As matrizes permitem combinar/modelar transformações geométricas

Processamento de vértices





Outros cálculos nesta fase:

- Material para cor
- Modelo de sombreamento
- Recorte

Outros cálculos nesta fase:

- Operação de *culling* do polígono
- Modo de renderização do polígono

Obs.: Ainda a acrescentar a matriz de textura

Operações com matrizes

Especificação da pilha de matrizes corrente

- gl.glMatrixMode(GL.GL_MODELVIEW or GL.GL_PROJECTION)

Outras operações com matrizes ou pilha de matrizes

- gl.glLoadIdentity() gl.glPushMatrix() gl.glPopMatrix()

Visor

- É usual utilizar a mesma dimensão da janela
- A razão de aspectos janela/visor de modo deve ser tida em consideração de modo a não distorcer a imagem projectada
- gl.glViewport(x, y, width, height)

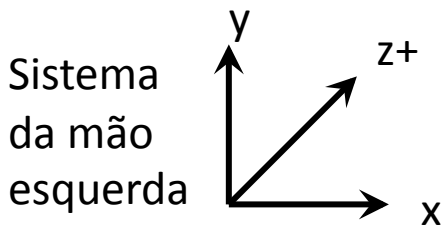
Transformações de modelação

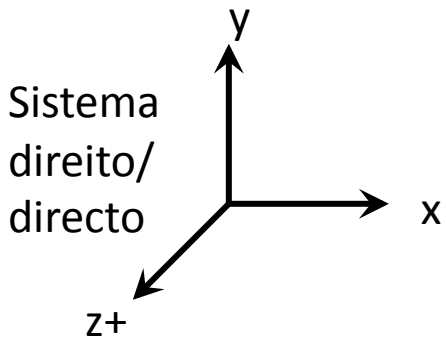
- Deslocação do objecto: gl.glTranslate{fd}(*x, y, z*)
- Rotação do objecto em torno de um eixo genérico: gl.glRotate{fd}(*angle, x, y, z*), com *angle* em graus
- Alteração de escala (ampliar ou reduzir) ou refexão do objecto: gl.glScale{fd}(*x, y, z*)

Transformações de projecção

As transformações de projecção (perspectiva e ortogonal) utilizam um sistema coordenado esquerdo

- Imaginar *zNear* e *zFar* como distâncias do ponto de visão
- Obs.: O remanescente em OpenGL é definido num sistema directo, incluindo a renderização de vértices



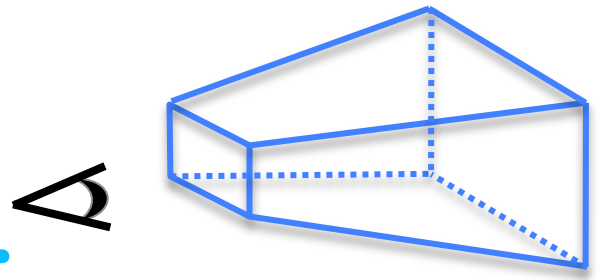


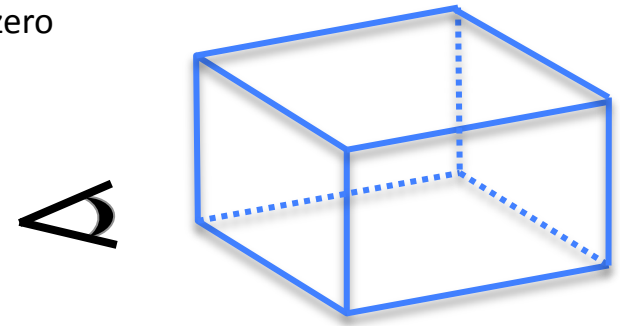
Projecção perspectiva

- glu.gluPerspective(*fovy, aspect, zNear, zFar*)
- gl.glFrustum(*left, right, bottom, top, zNear, zFar*)

Projecção paralela ortogonal

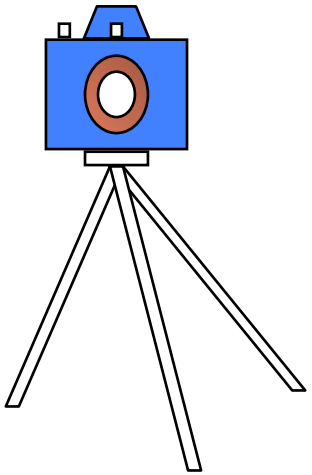
- gl.glOrtho(*left, right, bottom, top, zNear, zFar*)
- glu.gluOrtho2D(*left, right, bottom, top*)
  - Chamada de gl.glOrtho com valores de z próximos de zero





Transformações de visualização

Para "sobrevoar" a cena, deve-se alterar a transformação de visualização (posicionamento no espaço da máquina fotográfica/olho) e redesenhar a cena



- glu.gluLookAt(eye_x, eye_y, eye_z,
  aim_x, aim_y, aim_z,
  up_x, up_y, up_z)
- O vector up determina uma orientação única
- É preciso ter cuidado com posições "degeneradas"
- Através da sua manipulação, é possível criar modos próprios de perspectiva (e.g. polar view, pilot view)

Ligação entre visualização e modelação

- Deslocar a máquina é equivalente a deslocar o objecto definido em coordenadas do mundo real em direcção a uma máquina estática
- As transformações de visualização são equivalentes a várias transformações de modelação

Exemplo de alteração da dimensão da janela


```
public void reshape(GLAutoDrawable gLDrawable, int x, int y, int width, int height) {
  GL gl = gLDrawable.getGL();
  GLU glu = new GLU();
  gl.glViewport( 0, 0, width, height );
  gl.glMatrixMode( GL.GL_PROJECTION );
  gl.LoadIdentity();
  glu.gluPerspective( 65.0, (double) width / height, 1.0, 100.0 );
  gl.glMatrixMode(GL. GL_MODELVIEW );
  gl.glLoadIdentity();
  glu.gluLookAt( 0.0, 0.0, 5.0,  0.0, 0.0, 0.0, 0.0, 1.0, 0.0 );
}
```

gluPerspective e gluLookAt

É equivalente a

gluPerspective e glTranslate

Neste caso, a última linha do código acima seria substituída por

```
  gl.glTranslatef( 0.0, 0.0, -5.0 );
```

```
public void reshape(GLAutoDrawable gLDrawable, int x, int y, int width, int height) {
  GL gl = gLDrawable.getGL();
  GLU glu = new GLU();
  double aspect = (double) width / height;
  double left = -2.5, right = 2.5;
  double bottom = -2.5, top = 2.5;
  gl.glViewport( 0, 0, width, height );
  gl.glMatrixMode( GL_PROJECTION );
  gl.glLoadIdentity();
  if ( aspect < 1.0 ) {
    left /= aspect;
    right /= aspect;
  } else {
    bottom *= aspect;
    top *= aspect;
  }
  gl.glOrtho( left, right, bottom, top, near, far );
  gl.glMatrixMode( GL_MODELVIEW );
  gl.glLoadIdentity();
}
```

glOrtho

Composição de transformações de modelação

Problema: objectos hierárquicos

Uma posição depende da anterior (e.g: braço e mão de um robot; sub-componentes)

Solução: deslocação do sistema coordenado local

Transformações de modelação deslocam o sistema coordenado	Pós-mutiplicação de matrizes por colunas	Pós-multiplicação de matrizes em OpenGL

Operações inversas entre espaços de ecrã e do mundo real

Obs.: Em C

```
glGetIntegerv( GL_VIEWPORT, GLint viewport[4] )

glGetDoublev( GL_MODELVIEW_MATRIX, GLdouble mvmatrix[16] )

glGetDoublev( GL_PROJECTION_MATRIX, GLdouble projmatrix[16] )

gluUnProject( GLdouble winx, winy, winz,
                    mvmatrix[16], projmatrix[16],
                    GLint viewport[4],
                    GLdouble *objx, *objy, *objz )

gluProject( GLdouble objX,   GLdouble objY,   GLdouble objZ,
                    const GLdouble *model,
                    const GLdouble *proj,
                    const GLint*view,
                    GLdouble* winX,   GLdouble* winY,   GLdouble* winZ )
```

Problema: deslocamento dos objectos relativamente à origem do sistema coordenado absoluto

Objecto a rodar em torno de origem incorrecta (e.g. rodar em torno do seu centro)

Solução: sistema coordenado fixo

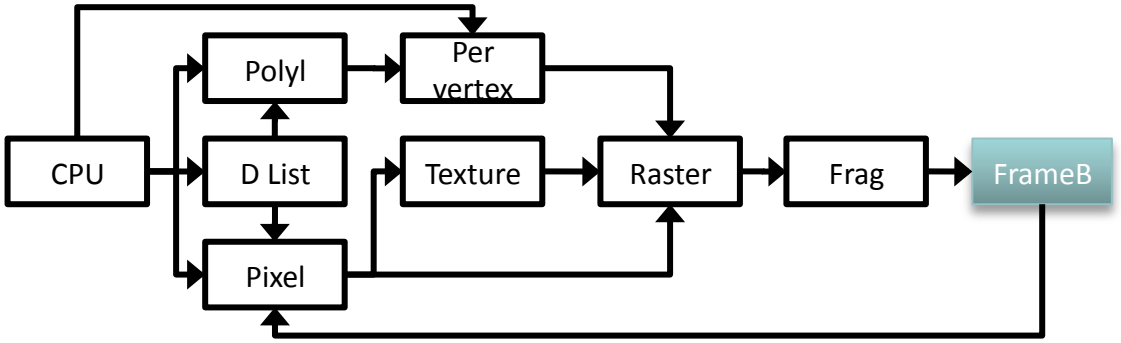
Transformações de modelação deslocam os objectos em torno de um sistema coordenado fixo	Pós-mutiplicação de matrizes por colunas	Pós-multiplicação de matrizes em OpenGL	É necessário inverter a ordem das operações para obter o resultado desejado

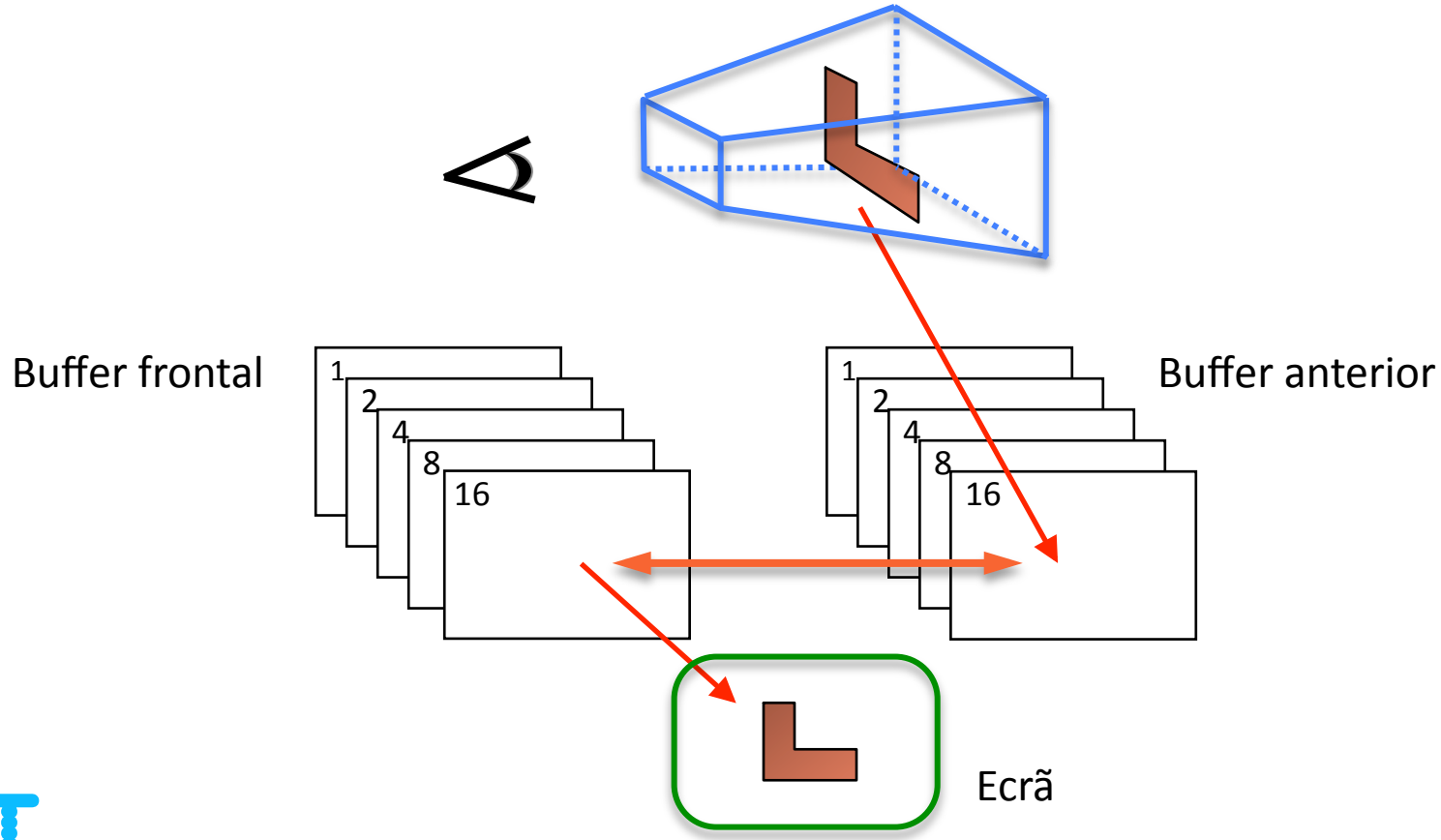
Sumário

- Arquitectura básica
- Renderização de primitivas
- Transformações
- Animação e profundidade
-
-
-

Buffer de cores duplo

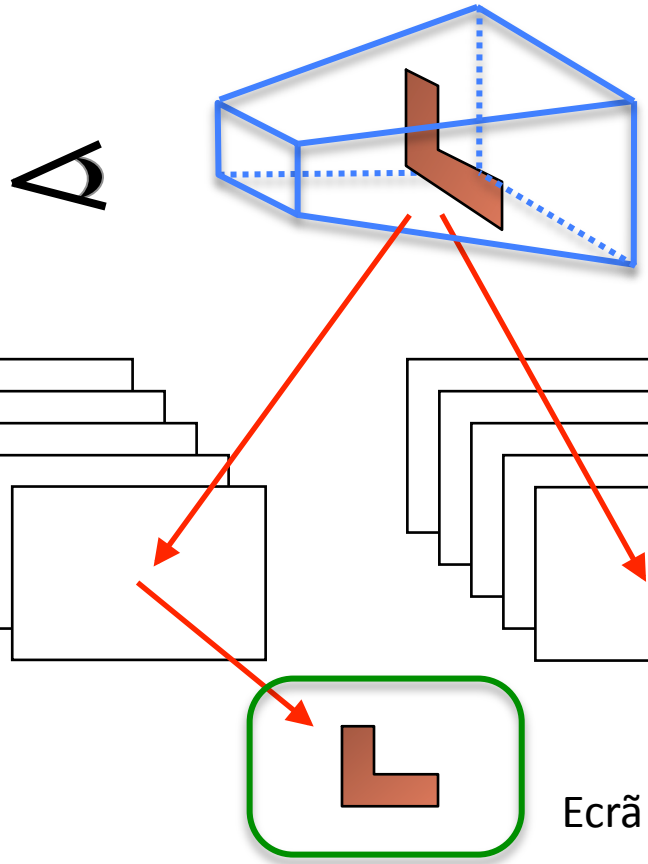
Troca de buffers: mostra um enquanto faz a renderização no outro



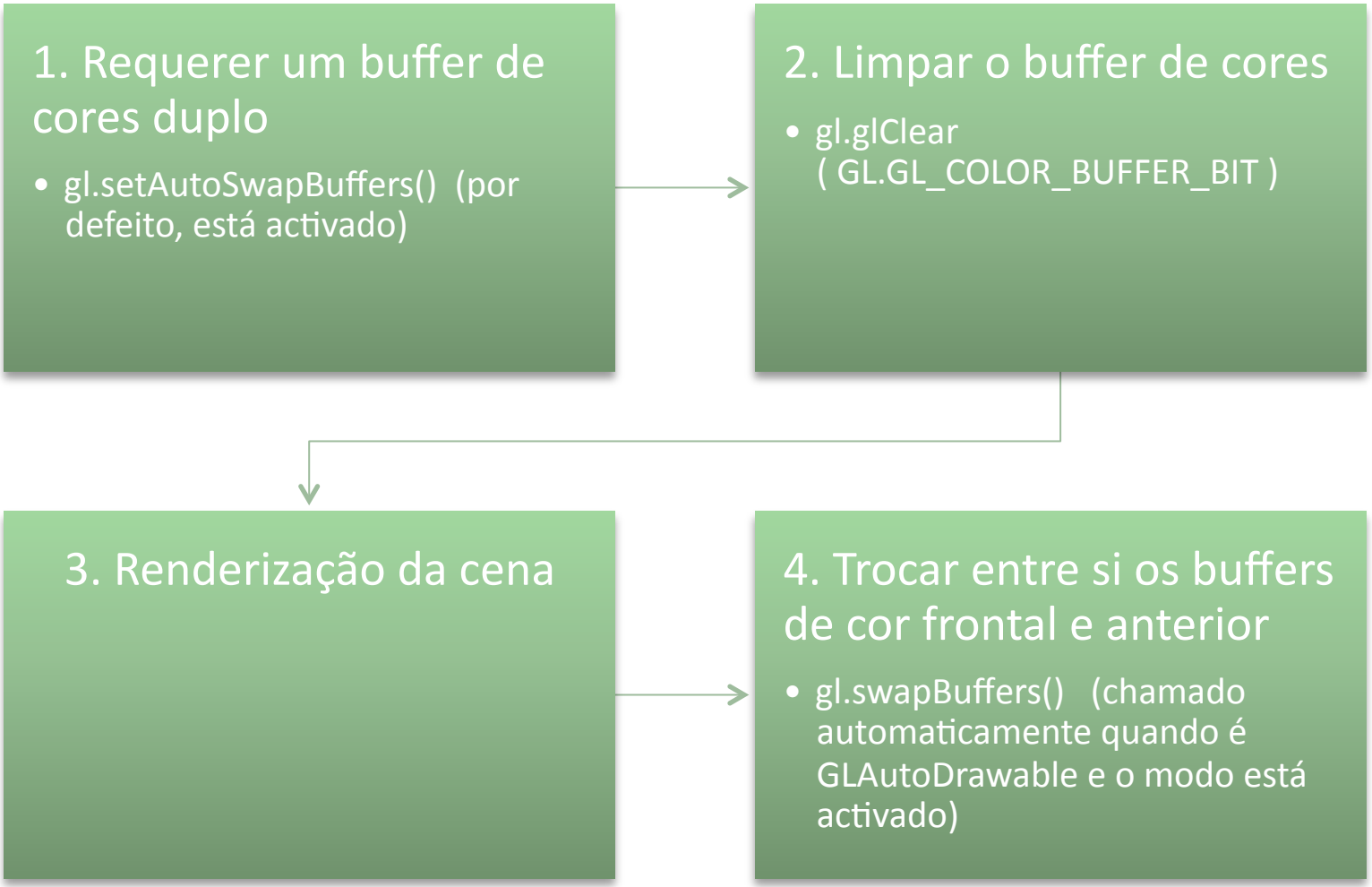


Buffer de profundidade

Para determinação de quais as primitivas que ocultam outras, através do algoritmo clássico de Z-buffer

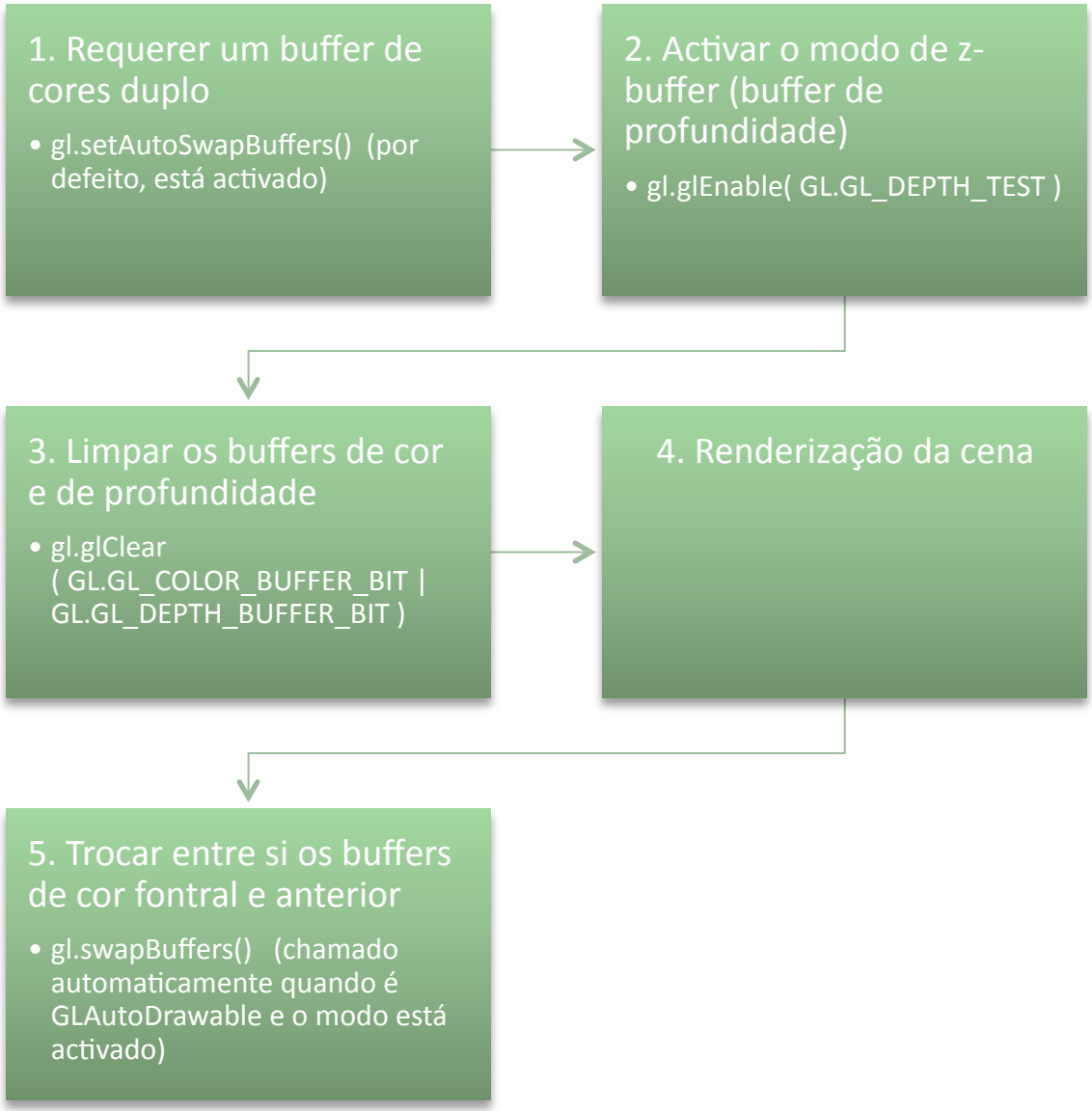


Animação com buffer de cores duplo



A animação obtém-se repetindo os passos 2, 3 e 4. Em JOGL, é usual utilizar-se um objecto da classe Animator ligado a um objecto da classe GLDrawable

Animação com buffer de cores duplo e buffer de profundidade



Exemplo: Moinho com a pá em movimento

```
import java.awt.Frame;
import java.awt.event.WindowAdapter; import java.awt.event.WindowEvent;
import java.awt.event.KeyEvent; import java.awt.event.KeyListener;
import javax.media.opengl.GL; import javax.media.opengl.GLAutoDrawable;
import javax.media.opengl.GLCanvas; import javax.media.opengl.GLEventListener;
import com.sun.opengl.util.FPSAnimator; import javax.media.opengl.glu.GLU;

public class Moinho implements GLEventListener, KeyListener {

   private boolean animado = true;
   private double alfa = 0;
   private static final double XYMAX = 100; // initial square window
   private static final int WH = 500;

   public void keyTyped(KeyEvent e) {
        if (e.getKeyChar() == 's') { animado = !animado; }
   }

  public void keyPressed(KeyEvent e) {}

  public void keyReleased(KeyEvent e) {}
```

```
public void init(GLAutoDrawable drawable) {
  GL gl = drawable.getGL();
  // setSwapInterval() provides a platform-independent way  to specify the minimum swap
  // interval for buffer swaps; an argument of 1 causes the application to wait for
  // the next vertical refresh until swapping buffers:
  gl.setSwapInterval(1);
  gl.glClearColor(0.0f, 0.0f, 0.5f, 0.0f);
  drawable.addKeyListener(this);
  System.out.println("\n*** Press 's' to stop or resume the animation ***\n");
 }

public void reshape(GLAutoDrawable drawable, int x, int y, int width, int height) {
  GL gl = drawable.getGL();
  GLU glu = new GLU();
  if (height <= 0)  height = 1;       // to avoid a divide by zero error!
  double aspecto = (double) width / (double) height;
  gl.glViewport(0, 0, width, height);
  gl.glMatrixMode(GL.GL_PROJECTION);
  gl.glLoadIdentity();
  if (aspecto < 1)   // initial aspect ratio = 1
      glu.gluOrtho2D(-XYMAX, XYMAX, -XYMAX / aspecto, XYMAX / aspecto);
  else
    glu.gluOrtho2D(-XYMAX * aspecto, XYMAX * aspecto, -XYMAX, XYMAX);
 }
```

```
public void display(GLAutoDrawable drawable) {
  GL gl = drawable.getGL();
  gl.glClear(GL.GL_COLOR_BUFFER_BIT);
  final double R = 60.0;
  double x0, y0, x, y;                int i;
  boolean ehVela = false;

  gl.glPolygonMode(GL.GL_FRONT_AND_BACK, GL.GL_FILL);
  gl.glMatrixMode(GL.GL_MODELVIEW);
  gl.glLoadIdentity();

  // …. inclusão de código para primitivas e.g cor, triângulos, quads, etc.

  gl.glPolygonMode(GL.GL_FRONT, GL.GL_FILL);  // default: counterclockwise orientation
  gl.glPolygonMode(GL.GL_BACK, GL.GL_LINE);   // default: clockwise orientation

  // … inclusão de código para as primitivas com animação, em função de um ângulo alfa

   if(animado) {
      alfa += 1.0;
      if (alfa > 360.0)    alfa -= 360.0;
   }
}

public void displayChanged(GLAutoDrawable drawable, boolean modeChanged, boolean
                           deviceChanged) {}
```

```
public static void main(String[] args) {
    Frame frame = new Frame("[CGI] Moinho");
    GLCanvas canvas = new GLCanvas();
    canvas.addGLEventListener(new Moinho());
    canvas.setSize(WH, WH);
    frame.add(canvas);
    frame.pack();
    // An Animator can be attached to one or more GLAutoDrawables to drive their display()
    // methods in a loop:
    final FPSAnimator animator = new FPSAnimator(canvas, 60); // 60 fps
    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            // Run this on another thread than the AWT event queue to make sure the call to
            // Animator.stop() completes before exiting
            new Thread(new Runnable() {
                public void run() {
                    animator.stop();      System.exit(0);
                }
            }).start();
        }
    });
    frame.setVisible(true);
    animator.start();
  }
}
```