Computação Gráfica (componente prática) - Parte 1

Temática: ***Introdução ao JOGL***

Actividade 1: Familiarização com o openGL em JAVA (JOGL) e implementação de algoritmos básicos de desenho gráfico

Competências a desenvolver:

- Compreender a estrutura básica de um programa em JOGL
- Aprender a instalar e configurar o ambiente Eclipse para trabalhar com JOGL
- Aprender a desenhar pontos e implementar algoritmos (do tipo *scan converting* em 2D) para desenho de segmentos de reta e caracteres
- Aprender as funcionalidades OpenGL incorporadas para a conversão scan (para segmentos de reta e polígonos)

Bibliotecas Gráficas

Uma biblioteca gráfica faculta uma série de comandos e funções que permitem criar programas gráficos em 2D e 3D. A biblioteca é incorporada em programas desenvolvidos em diferentes linguagens de programação (C, C++, Java, etc.) e podem correr em diversos tipos de plataformas e sistemas operativos. Os comandos podem especificar por exemplo, primitivas 2D e modelos geométricos 3D digitalizados, para serem visualizados. O termo "primitiva" significa que apenas algumas formas simples podem ser aceites pela biblioteca gráfica (como, pontos, segmentos de recta ou polígonos). Para desenhar formas complexas, é necessário o desenvolvimento de programas que combinam e agrupam as primitivas.

O OpenGL é a biblioteca gráfica que será integrada com o Java para introduzir a programação em computação gráfica.

Programação OpenGL em Java: JOGL

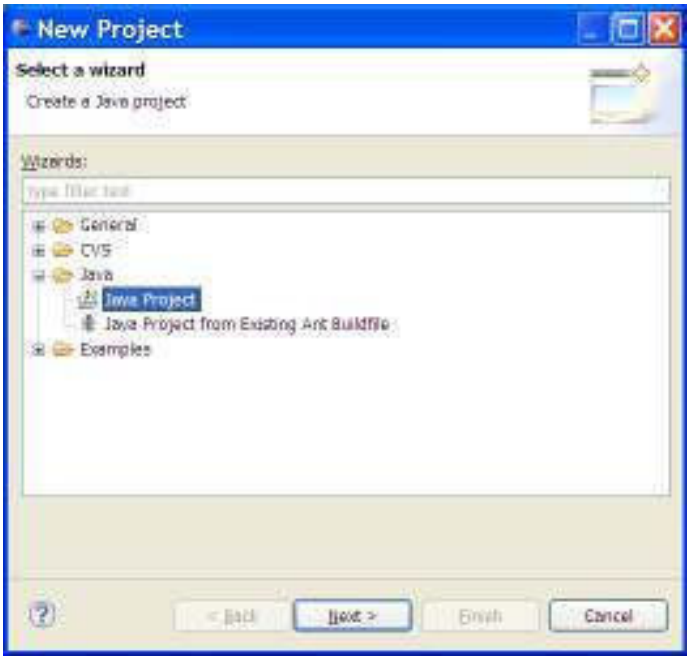
A OpenGL é uma das bibliotecas ou APIs gráficas mais difundida. Ela é suportada por diversas plataformas, garantindo o sucesso na passagem a produção da aplicação gráfica desenvolvida. O JOGL é uma das bibliotecas que permite o acesso total as funcionalidades da OpenGL, além de integrar as componentes do Java para desenvolvimento de interfaces gráficas (SWING e AWT).

A configuração e instalação do ambiente para desenvolvimento, deve incluir as seguintes etapas:

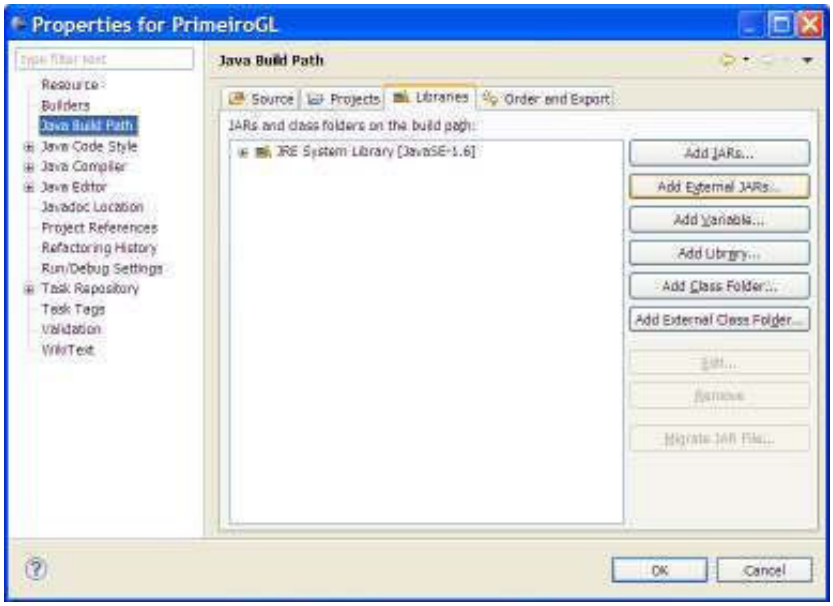
1. Baixar e instalar o JDK do Java (Java Development Kit) que contem um compilador, interpretador e debulhador para Java (http://www.oracle.com/technetwork/pt/java/javase/downloads/index.html)
2. Baixar e instalar o JOGL
3. Baixar e instalar um IDE para o Java (por exemplo, o Eclipse: http://www.eclipse.org/downloads/)

Após ter feito essas etapas, poderá criar o seu primeiro projeto com JOGL. Se for no Eclipse, as seguintes etapas devem ser seguidas:

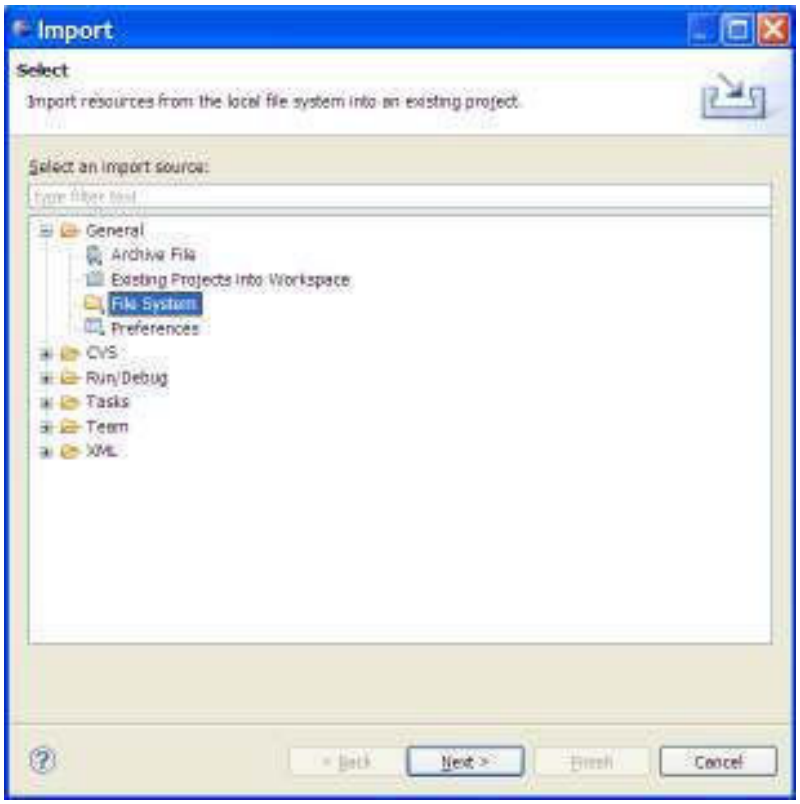
1. Crie um novo projeto, como um Java Project, utilizando as opções *default* do Eclipse



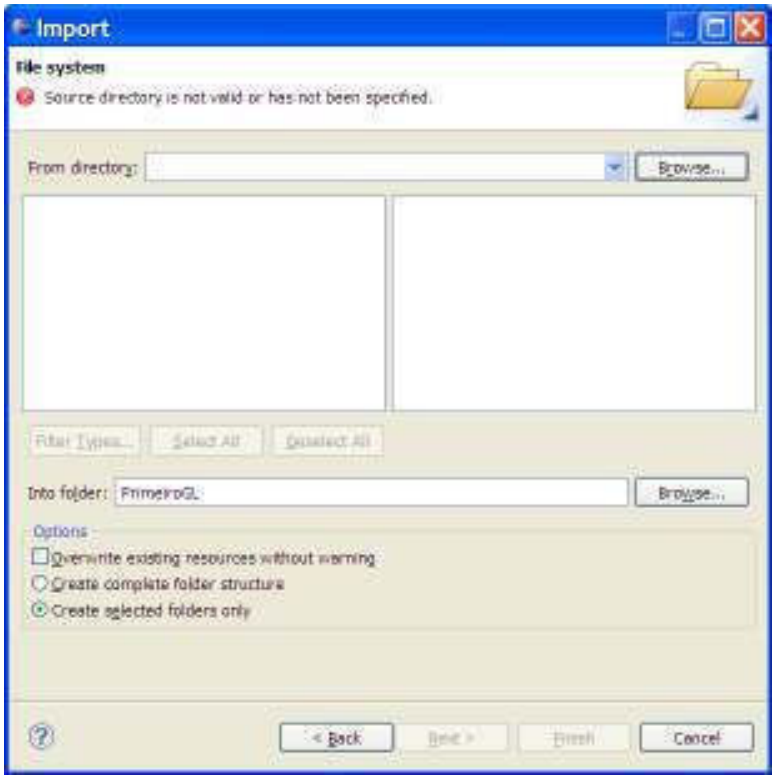
2. Adicione as bibliotecas gluegen-rt.jar e jogl.jar na opção *Add External JARs*. Estas bibliotecas estão na pasta para onde você descompactou o ficheiro de instalação do JOGL.



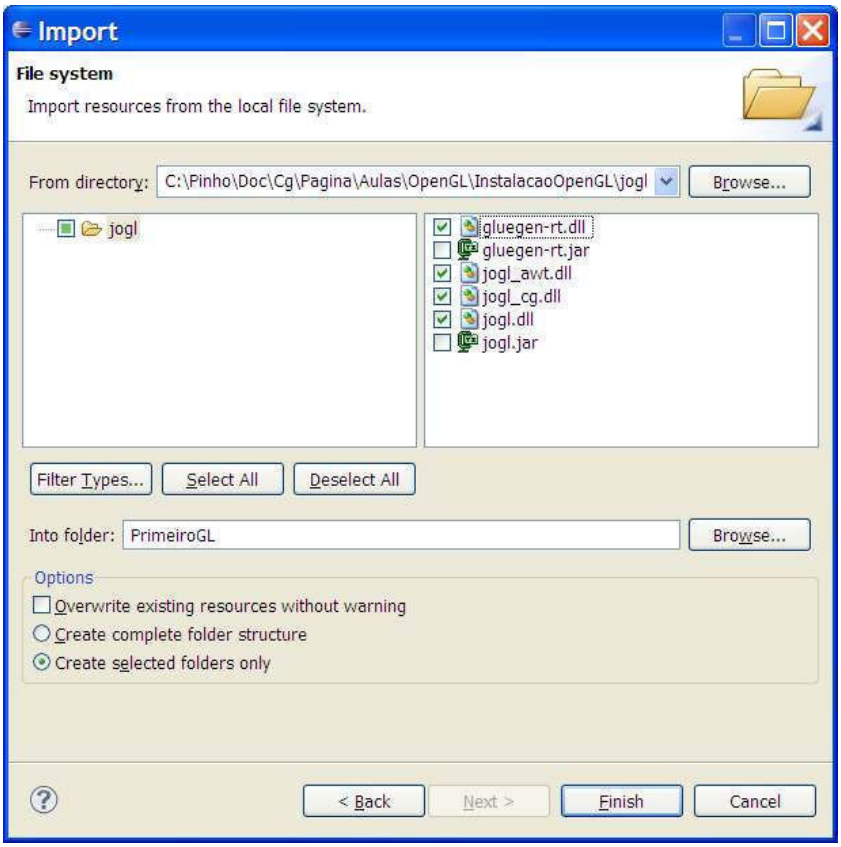
3. Incluir as DLLs gluegen-rt.dll, jogl.dll, jogl_awt.dll, jogl_cg.dll no projeto. Estes ficheiros estão na pasta para onde você descompactou o ficheiro de instalação do JOGL. No Eclipse, aceda *Menu File > Opção import > Item general >File system*



4. Clique no Botão *Next* e a seguir, clicando do Botão *Browse*, informe o nome da pasta onde estão as DLLs.



5. Selecione as DLLs e clique no botão *Finish*



Neste ponto a instalação da JOGL no projeto em Eclipse está completa. Para prosseguir, vamos criar o nosso primeiro programa para desenhar um ponto. Crie uma classe de nome "J1_0_Point" com o auxílio do menu *File* > *New* > *Class*.

Estrutura básica de um programa com o JOGL

Um programa gráfico apresenta sempre uma estrutura própria. O código abaixo exemplifica a criação de uma janela, de fundo preto, onde aparece apenas 1 ponto (com a activação de vários pixels) visível. O código está comentado de forma a esclarecer cada uma das suas partes:

```
// Inclusão das bibliotecas do Java para gestão de eventos e janelas
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

// Inclusão das bibliotecas do JOGL
import javax.media.opengl.*;

// A classe é pública e herda tudo da classe Frame (do Java) e implementa a
// interface GLEventListener (do JOGL)
public class J1_0_Point extends Frame implements GLEventListener {

	static int HEIGHT = 800, WIDTH = 800;
	static GL gl; // interface para o OpenGL
	static GLCanvas canvas; // uma frame desenhável
	static GLCapabilities capabilities;
```

```
public J1_0_Point() {

// 1. Especifica uma área na janela para desenho
	capabilities = new GLCapabilities();
	canvas = new GLCanvas();

// 2. Ativa a monitorização de eventos: redireccionados para: reshape
	canvas.addGLEventListener(this);

// 3. Adiciona o canvas ao contentor da Frame da janela
	add(canvas, BorderLayout.CENTER);

// 4. Interface para as funções do OpenGL
	gl = canvas.getGL();

// 5. Ativa a saída e fecho da janela ao clicar o botão X
	addWindowListener(new WindowAdapter() {
		public void windowClosing(WindowEvent e) {
			System.exit(0);
		}
	});
}

public static void main(String[] args) {
	J1_0_Point frame = new J1_0_Point();

// 6. Define o tamanho da frame e o torno visível
	frame.setSize(WIDTH, HEIGHT);
	frame.setVisible(true);
}

// É executada apenas 1x ao arrancar o programa
public void init(GLAutoDrawable drawable) {

// 7. Especifica a cor para desenho (RGB): branco
	gl.glColor3f(1.0f, 1.0f, 1.0f);
}

// Método chamado automaticamente sempre que ocorre um redimensionar
// da janela
public void reshape(GLAutoDrawable drawable, int x, int y, int width,
		int height) {

	WIDTH = width; // nova largura e altura são armazenados
	HEIGHT = height;

// 8. Especifica o tipo de projeção geométrica (sistemas de
// coordenadas) a utilizar na área de desenho (frame) coordenadas
	gl.glMatrixMode(GL.GL_PROJECTION);
	gl.glLoadIdentity();
	gl.glOrtho(0, width, 0, height, -1.0, 1.0);
	gl.glMatrixMode(GL.GL_MODELVIEW);
	gl.glLoadIdentity();

	gl.glViewport(100, 100, 100, 50);
}

// Método invocado constantemente para redesenho da área (pela reshape
```

```
        // indirectamente, ou ao mover a janela)
        public void display(GLAutoDrawable drawable) {

        // 9. Especificação do ponto a ser desenhado
              gl.glPointSize(10);
              gl.glBegin(GL.GL_POINTS);
              gl.glVertex2i(WIDTH / 2, HEIGHT / 2);
              gl.glEnd();
        }

        // Método invocado se ocorrer mudança de dispositivo ou de modo
        public void displayChanged(GLAutoDrawable drawable, boolean
modeChanged, boolean deviceChanged) {
        }
}
```

Explicação detalhada do código:

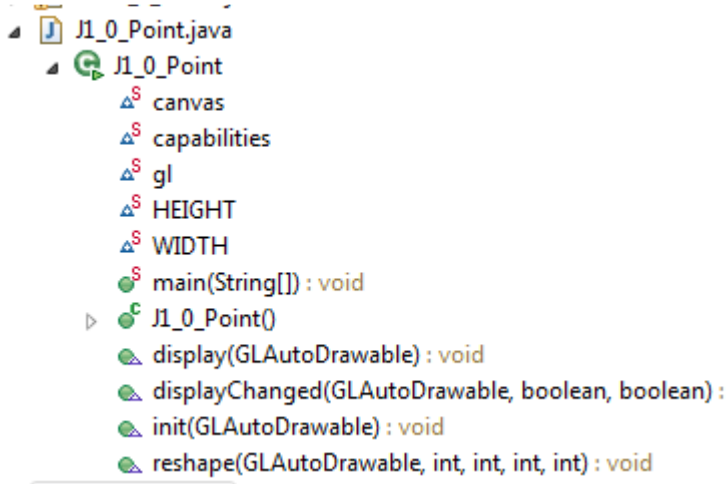
1. A classe ***GLCanvas*** é uma classe abstracta para a criação de janelas (AWT). Ela serve para suporte ao processo de renderização (desenho gráfico na janela) do OpenGL. O objecto ***canvas*** (que é do tipo ***GLCanvas***) corresponde a área de desenho que aparece dentro da moldura do objecto ***frame***, que corresponde a janela de visualização (que é do tipo ***Frame***, do Java).
2. O objecto ***frame*** é um *listener* para os eventos GL em ***canvas***. Quando algum evento ocorre, ele reencaminha ao gestor de eventos apropriado associado a ***canvas***. A classe-interface ***GLEventListener*** possui 4 métodos para gerir eventos:
   - ***init***() que é chamado imediatamente após a inicialização do OpenGL, sendo executada apenas uma vez. Nela se coloca normalmente todas as inicializações do OpenGL que devem ser executadas apenas no início;
   - ***reshape***() é invocado toda vez que o objecto ***canvas*** é redimensionado, isto é, quando o utilizador redimensiona a janela. O *listener* repassa para ela o objecto ***canvas***, o tamanho e canto inferior esquerdo (x, y) da área de visualização. Toda vez que este método é invocado, de seguida ele invoca o método ***display***();
   - ***display***() é invocado para inicializar o *rendering* do OpenGL. É chamado automaticamente sempre depois de um evento de redimensionamento da janela ocorrer;
   - ***displayChanged***() é invocado quando o modo ou dispositivo de visualização é alterado.
3. O ***canvas*** é adicionado ao ***frame*** de forma a cobrir toda a área de visualização. O ***canvas*** é redimensionado e redesenhado consoante o ***frame*** mude de dimensão.
4. ***gl*** é uma interface para gerir os métodos do OpenGL no JOGL. Por outro lado, todos os comandos do OpenGL têm como prefixo também "gl", logo, os comandos assumem uma forma como por exemplo ***gl.gl****Color*().
5. Para evitar o "pendurar" da janela, não conseguindo fecha-la, é adicionado um *listener* para o fecho da janela (clicar o botão X).
6. É definido o tamanho físico do objecto ***frame*** e o seu conteúdo é tornado visível. O tamanho físico depende da resolução do dispositivo e define o número de

pixels nas direcções x e y. Dependendo da versão do JOGL o tamanho pode considerar também as bordas da moldura da janela.

7. Estes métodos especificam as coordenadas lógicas. Por exemplo, se utilizarmos o comando ***glOrtho(0, largura, 0, altura, -1.0, 1.0)***, então as coordenadas do ***frame*** (ou ***canvas***) serão 0 <= x <= largura a partir do lado esquerdo até o lado direito da janela, 0 <= y <= altura a partir da base da janela até o seu topo e de -1 <= z <= 1 na direcção perpendicular a superfície da janela. A direcção z é ignorada nas aplicações 2D.
8. Estes métodos desenham um ponto em (largura/2, altura/2). As coordenadas são coordenadas lógicas não directamente relacionadas com o tamanho do ***canvas***. A largura e altura em ***glOrtho***() são o tamanho real da janela. Inicialmente, o valor da largura e altura têm o mesmo valor da janela, mas ao ocorrer um redesenho desta, eles assumem valores diferentes. Por isso, o ponto se move quando se redimensiona a janela.

Em suma, quando ***Frame*** é instanciado, o construtor de ***J1_0_Point()*** cria uma área de desenho, associa um monitor de eventos, um visualizador, e captura um *handle* para os métodos ***gl***. ***reshape***() define as coordenadas lógicas na moldura da janela. ***display***() desenha o ponto nas coordenadas lógicas. Quando o programa inicia, ***main***() é invocado automaticamente, e então, ***frame*** é instanciado e de seguida***, J1_0_Point***(), ***setSize***(), ***setVisible***(), ***init***(), ***reshape***(), e ***display***(). ***reshape***() e ***display***() são invocados várias vezes caso o utilizador mexa na área de visualização.

A imagem abaixo mostra a estrutura final da classe dentro do projeto Java no Eclipse.



Para compilar e executar o programa, basta ir no menu *Run* > *Run as* > *Java Application*. O botão verde com uma seta é um atalho para este comando. Você pode encontrar manuais para ajudar a compreender as funcionalidades disponíveis no Eclipse (http://www.eclipse.org/documentation/).

No seguimento deste exemplo de código, está disponível na plataforma o programa ***J1_1_Point***. A classe ***J1_0_Point*** utiliza o mecanismo de herança para herdar tudo o que foi definido para a ***classe J1_0_Point*** e adiciona outras funcionalidades disponíveis no OpenGL. Este programa serve para desenhar pontos aleatoriamente no ecrã.

Crie um novo projeto Java, seguindo os passos anteriores, e adicione uma classe com o nome ***J1_1_Point*** (o Java exige que o nome do ficheiro da classe seja sempre igual ao da classe que se declara nele). Como no caso anterior, segue-se uma explanação mais pormenorizada dos comentários inseridos no código:

1. Como ***J1_1_Point*** é construída a partir da classe anterior, o construtor desta última é automaticamente chamada ao iniciar a execução do programa. Ativamos a utilização de um único *buffer* para a gestão do desenho do ecrã(pode ser duplo, como se verá mais a frente).
2. ***glClearColor***() especifica a cor de fundo para limpar e pintar com ***glClear***(). Como o OpenGL é uma máquina de estados, essa cor fica ativa até que se mande alterar.
3. O objecto ***animator*** é associado ao objecto ***canvas*** de forma a criar um *thread* que coloca a execução do método ***display***() num laço de repetição, criando um efeito de animação. O *thread* é um processo que corre paralelamente com a execução do programa. Como o Java suporta o multi-processo, o ***animator*** só é destruído no fecho da janela.
4. É gerado um ponto aleatório. Graças ao objecto ***animator*** causar a repetição seguida do método ***display***() dentro do *thread* que criou.

Em suma, o construtor da classe mãe é chamado implicitamente, criando o objecto ***canvas***, adicionando o *listener* e associando o método ***display***() a ele. ***reshape***() definirá as coordenadas lógicas da janela de visualização. ***animator.start***() invoca seguidamente display() a partir da *thread*. ***display***() desenha um ponto nas coordenadas lógicas. Quando o programa inicia, ***main***() é executado, e em seguida vários pontos vermelhos aparecem na janela.

Conversão scan (por varredura)

Segmentos de reta

O programa ***J1_2_Line*** está disponível na plataforma e ilustra a utilização do JOGL para o desenho de segmentos de reta com o algoritmo *scan line*. Este algoritmo se baseia na equação da reta e ativa os pixels mais apropriados para representar visualmente a reta pretendida. A maior ou menor perfeição da reta, depende do arredondamento que se considere no cálculo dos pixels (já que o ecrã é composto por uma matriz de pixels). Toda a reta para ser desenhada no ecrã tem que ser convertida por um algoritmo deste tipo.

A classe ***J1_2_line*** é uma subclasse de ***J1_1_point***, herdando portanto todos os seus métodos e atributos públicos. O construtor, ***init***(), ***reshape***(), e alguns outros métodos são todos herdados. Por exemplo, na inicialização "*J1_2_line f = new J1_2_line();*", o construtor de ***J1_1_point*** é invocado, que por sua vez invocará o construtor de ***J1_0_Point***, inicializando o ***canvas***, o *handle* ***gl***, e etc. Depois das inicializações, a versão do método ***init***() em ***J1_1_point*** é invocado, pois foi executada um *override* dos métodos herdados.

O programa ***J1_3_Line*** também está disponível na plataforma e ilustra a implementação do algoritmo de Bresenham para o desenho de segmentos de reta em JOGL. A classe ***J1_3_Line*** é criada como filha da anterior. Embora estejamos a desenhar com auxílio do algoritmo de Bresenham, o OpenGL possui função própria para executar a conversão *scan* do segmento de reta que lhe é passado:

```
gl.glBegin(GL.GL_LINES);
      gl.glVertex2i(x0,y0);
      gl.glVertex2i(xn,yn);
glEnd();
```

Embora o programa ***J1_3_line.java*** seja somente uma simulação, pois não manipula directamente o *frame buffer*, ele ajuda a compreender como o processo de conversão *scan* é executado.

Curvas, triângulos e polígonos

Como todas as formas podem ser representadas por segmentos de reta, isto significa que é possível se desenvolver um algoritmo genérico para executar a conversão scan de qualquer forma. Este conceito é importante pois muitas outras operações gráficas são executadas tendo como base a análise *scan* do objecto a ser desenhado (importante na eliminação das superfícies escondidas, p. e.).

Uma biblioteca gráfica fornece funções básicas para o desenho de primitivas. Por exemplo, o OpenGL desenha polígonos convexos com os seguintes comandos:

```
gl.glBegin(GL.GL_POLYGON);
      gl.glVertex2i(x0,y0);
      . . .  // Lista de vértices
      gl.glVertex2i(xn,yn);
glEnd();
```

Um polígono convexo significa que todos os ângulos internos do polígono formados pelas arestas são inferiores a 180º. Se ele não for convexo, é côncavo. Os polígonos convexos são mais rapidamente convertidos via *scan* que os côncavos.

Os programas ***J1_3_Triangle e J1_3_CircleLine*** ilustram a utilização do algoritmo de Bresenham para desenhar um triângulo e círculo, respectivamente. Os códigos estão na plataforma, e como os demais, estão comentados.

Em resumo, existem diferentes algoritmos para a conversão scan de primitivas gráficas (retas, polígonos, etc.). Se uma função para conversão scan de uma primitiva não é facultada pela biblioteca gráfica, é possível criar ou implementar uma já existente.

Cadeia de caracteres

Os caracteres são polígonos. Entretanto, eles são tão frequentemente utilizados que são usualmente armazenados numa biblioteca de fontes de letras. No caso do JOGL, através da classe GLUT podemos aceder um conjunto de fontes do tipo bitmap e *stroke* (por traços) independentes da plataforma. Elas podem ser utilizadas em 3D. ***glutBitmapCharacter***() desenha um caracter bitmap na posição indicada. A posição *raster* é um ponto (x, y, z) no volume de visualização, que é especificado através de ***glRaster***(x, y, z). ***glutBitmapString***() desenha uma cadeia de caracteres na posição *raster* actual. ***glutStrokeCharacter***() desenha uma cadeia de caracteres por traços na posição indicada. As fontes do tipo *stroke* são esboços simples de letras, que são transformadas (a transformação é tópico a ser abordado mais a frente) como objectos 3D.

O programa ***J1_3_xFont*** exemplifica a utilização das funções para caracteres disponíveis no OpenGL. A classe ***J1_3_xFont*** é filha da classe ***J1_3_Triangle***, e portanto, herda todos os métodos e atributos públicos desta.

Computação Gráfica (componente prática) - Parte 2

Temática: ***Utilização do frame buffer, do efeito anti-escada e da animação***

Actividade 2: Familiarização com a utilização do *frame buffer*, do efeito *antialiasing* e animação (2D) em computação gráfica utilizando o JOGL

Competências a desenvolver:

- Implementar o desenho de segmentos de reta (algoritmo *scan line*) incluindo o efeito *antialiasing* (algoritmo apropriado)
- Saber as diferenças existentes na utilização do *single* para o *double frame* buffer
- Conhecer as potencialidades da utilização do *double frame buffer* para criar efeitos animados

Atributos Visuais e Efeito Escada

Em geral, qualquer parâmetro que afecte a forma que uma primitiva é desenhada, é referido como atributo. Por exemplo, uma linha pode ter vários atributos como a cor, o brilho, o tipo (pontilhado, tracejado, etc.), etc. Por outro lado, o ecrã do computador e o seu *frame buffer* correspondente possuem tamanhos finitos. Por isso, qualquer segmento de reta, curva ou aresta de polígono apresentam o indesejável efeito escada (*aliasing*) quando desenhados no ecrã. Existem vários métodos para minorar esse efeito indesejável. A utilização e parametrização de certos atributos visuais nas primitivas podem ajudar muito a reduzir o efeito escada.

Desenho de retas com redução do efeito escada

O exemplo disponível no programa ***J1_4_Line.java*** ilustra a implementação do algoritmo de Bresenham, tendo em conta a redução do efeito escada (como você notará, todos os programas-exemplo facultados com este e demais apontamentos, atendem, sempre que possível o mecanismo de herança, reutilizando classes anteriores sempre que possível). No código, o método ***IntesifyPixel*** serve para calibrar o atributo de cor do pixel consoante a distância que este possua ao segmento de linha "ideal". A função ***writepixel(x, y, flag)*** escreve o pixel na posição quadrante consoante o valor da ***flag*** passada (ver o código ***J1_3_Line.java***). A cor do pixel é mais ou menos intensificada conforme a maior ou menor visibilidade desejada para o mesmo, com a função ***glColor3f(r1, g1, b1)***. Quando maior for a distância do pixel da reta "ideal", mas "esbatido" deverá ser o seu aspecto, enquanto os mais próximos, devem ficar mais "visíveis".

Double Buffer e Animação

Um efeito dinâmico pode ser obtido quando se projecta imagens a uma taxa de 24 quadros por Segundo. A animação num computador pode ser obtida da mesma forma, isto é, alternando-se várias imagens diferentes e de forma sequencial a uma dada velocidade. Neste caso, a taxa de refrescamento é dada pela velocidade com que o *frame buffer* é lido e carregado para o ecrã do computador pelo controlador de vídeo. Uma taxa de 60 quadros por segundo é mais suave visualmente do que uma de 30, embora a de 120 seja marginalmente melhor do que a de 60, por exemplo.

Considerando uma taxa de 60 quadros/segundo, um programa de animação poderia ser criado com a seguinte estrutura básica:

```
open_window_with_single_buffer_mode();
        for (i = 0; i < 100; i++) {
                clear_buffer();
                draw_frame(i);
                wait_until_1/60_of_a_second_is_over();
        }
```

Para resolver este problema, em lugar de um único *frame buffer*, podemos ter dois, ou seja, um ***double-buffering***. Neste caso, um dos *frame buffers* é designado de frontal enquanto o outro de posterior. No frontal fica a imagem que está sendo vista, enquanto no posterior está a ser desenhada ("escrita") a imagem seguinte, após a conversão *scan* desta. Quando o desenho fica completo, eles são trocados, isto é, o frontal passa a ser o posterior e vice-versa.

Um programa de animação mais aprimorado inclui normalmente a seguinte estrutura básica:

```
open_window_with_double_buffer_mode();
        for (i = 0; i < 100; i++) {
                clear_back_buffer();
                draw_frame_into_back_buffer(i);
                wait_until_1/60_of_a_second_is_over();
                swap_buffers();
        }
```

O JOGL utiliza o método ***capabilities.setDoubleBuffered(true)*** para activar o ***double buffer***. O método ***animator*** é responsável por executar o método display num laço "infinito" (ou melhor, até que haja uma interrupção da execução de ***animator*** – fecho de janela, alteração de cena, etc.). Quando o modo de *double buffer* está activado, este método alterna o desenho das cenas entre o buffer frontal e posterior de forma automática e por defeito. A desactivação do modo de *double buffer* é obtida com o método ***drawable.setAutoSwapBufferMode(false).*** Uma vez invocada, uma nova activação do modo *double buffer* deverá ser feita com a invocação "manual" (feita pelo programador em algum outro ponto do programa) com o método

O programa ***J1_5_Circle.java*** (disponível na plataforma) é um exemplo que demonstra a utilização da animação: desenha um círculo cujo raio que se alterna a cada quadro, utilizando o modo *double buffer* para melhorar a performance. O programa inclui algumas operações vectoriais básicas, bastante úteis no "universo" da computação

gráfica. O círculo é desenhado de forma aproximada, considerando-se que ele é composto por um conjunto de triângulos que são subdivididos de forma contínua. A figura 1 ilustra a situação:

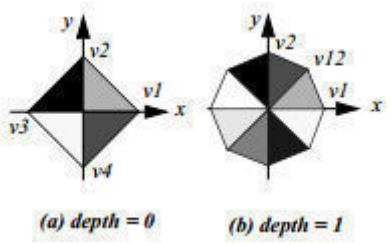


Figura 1

No início, são facultados os seguintes vértices (com coordenadas x, y e z): v1, v2, v3, v4 e o ponto central v(). Quando a variável ***depth = 0***, são desenhados apenas 4 triângulos e o círculo apresenta uma forma quadrangular. Quando essa variável passa para o valor 1, cada triângulo é subdividido em 2, e portanto, são desenhados um total de 8 triângulos. Considerando v1 e v2, como se pode calcular v12?

Se considerarmos os 3 como sendo vectores, então, v12 é o vector direcção definido por ***(v1 + v2) = (v1x+v2x, v1y+v2y, v1z+v2z)*** e os comprimentos dos vectores é dado por: ***|v1|*** = ***|v2|*** = ***|v12|***. Se o valor do raio for unitário, então ***v12 = normalize(v1 + v2).*** A normalização escalona um vector para o seu equivalente unitário. Em geral, ***v12 = cRadius*normalize(v1+ v2)***, e para cada quadro o valor de ***CRadiusTo*** é alterado de forma a obter um efeito animado.

Os demais vértices desconhecidos podem ser calculados tendo como base o mesmo raciocínio descrito acima – adição de vectores e respectiva normalização. A subdivisão avança mais ou menos consoante o valor atribuído a ***depth***, ou seja, o raio é aumentado ou diminuído repetidamente. Alguns pormenores a destacar no código:

1. Para o efeito animado é a activado o modo *double-buffering*.
2. O novo valor de ***w*** e ***h*** retornado por ***Drawable*** no método ***reshape***() são guardados na variáveis globais estáticas inteiras ***WIDTH*** e ***HEIGHT*** (foram declaradas em ***J1_0_Point.java***). Elas são utilizadas para definir novas coordenadas a cada interacção e controlar, mais tarde, a dimensão do raio do círculo.
3. As coordenadas para uma nova área de desenho são especificadas. A origem é no centro da área. Por isso, se a área for redimensionada, o centro é alterado em consonância, para uma nova localização.
4. O ***frame buffer*** é limpo a cada redesenho do círculo.
5. Como cada vértice do triângulo é diferente dos demais, cada um é pintado com uma cor correspondente a sua coordenada espacial (do vértice). Para evitar valores negativos para a cor, pois cada componente RGB é definido com valores numa escala entre 0 a 1, é utilizado sempre o quadrado de cada componente, x, y e z do vector normal.

Computação Gráfica (componente prática) - Parte 3

Temática: ***Transformações geométricas em 2D e 3D***

Actividade 3: Familiarização com o processo de transformação geométrica 2D e 3D com a utilização do JOGL

Competências a desenvolver:

- Aprender a implementar transformações 2D em objectos gráficos: Translação, rotação e alteração de escala
- Aprender a implementar transformações 3D em objectos gráficos: Translação, rotação e alteração de escala
- Compreender e executar transformações compostas (em 2D e 3D)

Simulação da Implementação em OpenGL

O OpenGL implementa efectivamente transformações 3D, que serão discutidas mais tarde. O programa ***J2_0_2DTransform.java*** ilustra a forma como o OpenGL implementa as transformações em 2D a nível do *hardware*. O OpenGL possui uma estrutura de dados designada de MODELVIEW, que nada mais é do que uma *stack* utilizada para armazenar as matrizes de transformação. No topo da pilha está sempre a última matriz de transformação. Consideremos uma *stack* de matrizes da seguinte forma:

Implementação da transformação 2D ao estilo OpenGL

```
import javax.media.opengl.*;
public class J2_0_2DTransform extends J1_5_Circle {
        private static float my2dMatStack[][][] = new float[24][3][3];
        private static int stackPtr = 0;
        ...
}
```

A matriz identidade para coordenadas homogéneas 2D é dada por $\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$. Qualquer matriz multiplicada pela matriz identidade, não se modifica.

O atributo ***stackPtr*** aponta para a matriz no topo (por consequência, corrente) da pilha de matrizes (***my2dMatrixStack[stackPtr]***). As transformações são obtidas com os seguintes métodos: ***my2dLoadIdentity()***, ***my2dMultMatrix(float mat[][])***, ***my2dTranslatef(float x, float y)***, ***my2dRotatef(float angle)***, ***my2dScalef(float x, float y)***, e ***my2dTransformf(float vertex[], float vertex1[])*** (ou ***my2dTransVertex(float vertex[], float vertex1[])*** para vértices já na forma homogénea).

1. ***my2dLoadIdentity()*** carrega a matriz corrente na pilha de matrizes com a sua identidade.

```
// Inicializa uma matriz 3 x 3 a zeros
private void my2dClearMatrix(float mat[][]) {
	for (int i = 0; i<3; i++) {
		for (int j = 0; j<3; j++) {
			mat[i][j] = 0.0f;
		}
	}
}

// Inicializa uma matriz como a Identidade
private void my2dIdentity(float mat[][]) {
	my2dClearMatrix(mat);
	for (int i = 0; i<3; i++) {
		mat[i][i] = 1.0f;
		}
}

// inicializa a matriz atual com a matriz Identidade
public void my2dLoadIdentity() {
	my2dIdentity(my2dMatStack[stackPtr]);
}
```

2. ***my2dMultMatrix(float mat[][])*** multiplica a matriz corrente na pilha de matrizes com a matriz ***Mat***: ***CurrentMatrix = currentMatrix*Mat***.

```
// Multipplica a matriz atual com a matriz Mat
public void my2dMultMatrix(float mat[][]) {
	float matTmp[][] = new float[3][3];
	my2dClearMatrix(matTmp);
	for (int i = 0; i<3; i++) {
		for (int j = 0; j<3; j++) {
			for (int k = 0; k<3; k++) {
				matTmp[i][j] += my2dMatStack[stackPtr][i][k]*mat[k][j];
			}
		}
	}
// Salva os resultados na matriz corrente
```

```
        for (int i = 0; i<3; i++) {
                for (int j = 0; j<3; j++) {
                        my2dMatStack[stackPtr][i][j] = matTmp[i][j];
                }
        }
}
```

3. ***my2dTranslatef(float x, float y)*** multiplica a matriz corrente na pilha de matrizes com matriz transformação ***T(x, y):***

```
// multiplica a matriz corrente com a matriz de transformação
public void my2dTranslatef(float x, float y) {
        float T[][] = new float[3][3];
        my2dIdentity(T);
        T[0][2] = x;
        T[1][2] = y;
        my2dMultMatrix(T);
}
```

4. ***my2dRotatef(float angle)*** multiplica matriz corrente na pilha de matrizes com a matriz de rotação ***R(angle)***:

```
// multiplica a matriz corrente com a matriz de rotação
public void my2dRotatef(float angle) {
        float R[][] = new float[3][3];
        my2dIdentity(R);
        R[0][0] = (float)Math.cos(angle);
        R[0][1] = (float)-Math.sin(angle);
        R[1][0] = (float)Math.sin(angle);
        R[1][1] = (float)Math.cos(angle);
        my2dMultMatrix(R);
}
```

5. ***my2dScalef(float x, float y)*** multiplica a matriz corrente na pilha de matrizes com a matriz de escala ***S(x, y):***

```
// multiplica a matriz corrente com a matriz escala
public void my2dScalef(float x, float y) {
        float S[][] = new float[3][3];
        my2dIdentity(S);
        S[0][0] = x;
        S[1][1] = y;
        my2dMultMatrix(S);
}
```

6. ***my2dTransformf(float vertex[]; vertex1[])*** multiplica a matriz corrente na pilha de matrizes com ***vertex***, e salva o resultado em ***vertex1***. ***vertex*** é inicialmente convertido para coordenadas homogéneas antes da multiplicação matricial.

```
// v1 = (a matriz actual) * v
// v e v1 são vértices em coordenadas homogéneas
public void my2dTransHomoVertex(float v[], float v1[]) {
	int i, j;
	for (i = 0; i<3; i++) {
		v1[i] = 0.0f;
	}
	for (i = 0; i<3; i++) {
		for (j = 0; j<3; j++) {
			v1[i] += my2dMatStack[stackPtr][i][j]*v[j];
		}
	}
}
// vertex = (a matriz actual) * vertex
// vertex está em coordenadas homogénas
public void my2dTransHomoVertex(float vertex[]) {
	float vertex1[] = new float[3];
	my2dTransHomoVertex(vertex, vertex1);
	for (int i = 0; i<3; i++) {
		vertex[i] = vertex1[i];
	}
}
// transforma v em v1 consoante a matriz transformação actual
//  v e v1 não estão em coordenadas homogéneas
public void my2dTransformf(float v[], float v1[]) {
	float vertex[] = new float[3];
// passa para coordenadas homogéneas
	vertex[0] = v[0];
	vertex[1] = v[1];
	vertex[2] = 1;
// multiplica vertex pelas matriz actual
	my2dTransHomoVertex(vertex);
// retorna para coordenadas em 3D
	v1[0] = vertex[0]/vertex[2];
	v1[1] = vertex[1]/vertex[2];
}
// transforma v consoante a matriz transformação actual
//  v não está em coordenadas homogéneas
public void my2dTransformf(float[] v) {
	float vertex[] = new float[3];
// passa para coordenadas homogéneas
	vertex[0] = v[0];
	vertex[1] = v[1];
	vertex[2] = 1;
// multiplica vertex pelas matriz actual
	my2dTransHomoVertex(vertex);
// retorna para coordenadas em 3D
```

```
        v[0] = vertex[0]/vertex[2];
        v[1] = vertex[1]/vertex[2];
}
```

7. Em adição aos métodos acima, ***my2dPushMatrix()*** e ***my2dPopMatrix()*** são mecanismos potentes para alterar a matriz corrente na pilha de matrizes (serão discutidos mais a frente). ***PushMatrix*** incrementa o ponteiro para a pilha e faz uma cópia da matriz anterior para a matriz corrente. Por isso, a matriz permanece a mesma, embora estejamos utilizando zonas de memória distintas na pilha de matrizes. ***PopMatrix*** decrementa o ponteiro para a pilha, de forma a termos como activa a matriz anterior que foi salva em ***PushMatrix***.

```
// move o ponteiro da pilha para cima e copia a matriz anterior
// matriz atualizada para a matriz corrente
public void my2dPushMatrix() {
        int tmp = stackPtr+1;
        for (int i = 0; i<3; i++) {
                for (int j = 0; j<3; j++) {
                        my2dMatStack[tmp][i][j] = my2dMatStack[stackPtr][i][j];
                }
        }
        stackPtr++;
}
// move o ponteiro para baixo
public void my2dPopMatrix() {
        stackPtr--;
}
```

Com o auxílio dos métodos descritos acima, o exemplo apresentado no código ***J2_0_2DTransform.java*** consegue executar diferentes transformações espaciais num triângulo.

Composição de Transformações

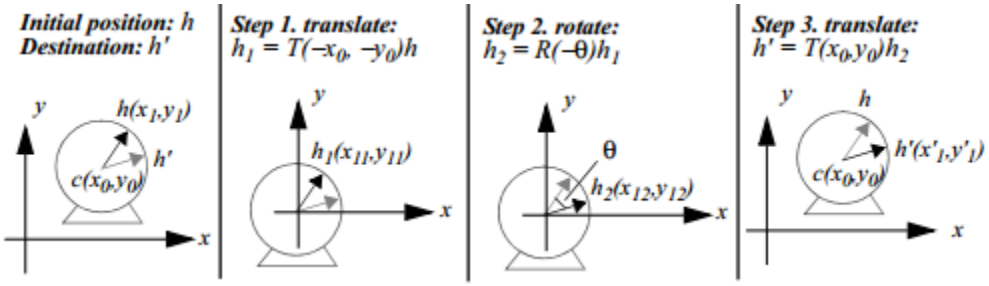
Uma transformação complexa e composta é normalmente obtida pela execução de uma sequência de transformações espaciais simples. O resultado é a combinação entre translações, rotações e alterações e escala.

Consideremos o seguinte exemplo: Calcular as coordenadas espaciais dos ponteiros de um relógio em 2D após se moverem. Consideremos apenas um dos ponteiros. O centro de rotação é dado por $c(x_0, y_0)$, e a posição final de rotação está em $h(x_1,y_1)$. Se soubermos o ângulo de rotação, conseguiremos saber a nova posição h' depois da rotação? Sim, pois conseguimos calcular com a concatenação de várias transformações espaciais.

1. Mova o ponteiro de forma ao centro de rotação se situar na origem do sistema de coordenadas

2. Gire θ graus em torno da origem o ponteiro. Note que a direcção positiva de rotação é no sentido dos ponteiros do relógio.
3. Depois da rotação, devolva o relógio a sua posição original

A figura abaixo ilustra a situação.



Em termos práticos, isto corresponderá a multiplicação, na seguinte ordem, das matrizes de translação em (1), rotação em (2) e translação em (3). A ordem que a multiplicação é executada é importante, pois produz resultados diferentes.

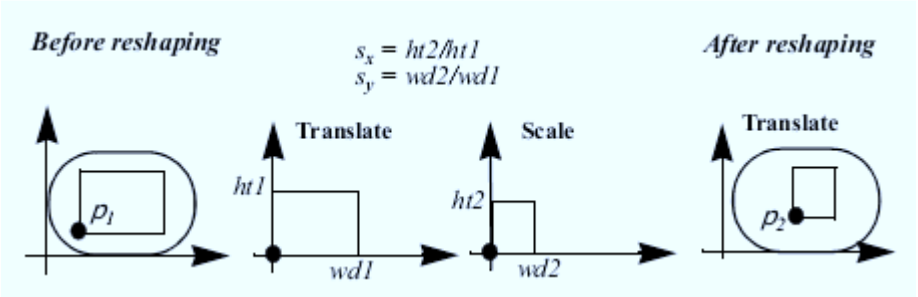
No estilo actual de implementação do OpenGL, a multiplicação de matrizes é feita da esquerda para direita, e existe sempre uma matriz final na pilha de matrizes. O segmento seguinte do código ***J2_1_Clock2d.java*** ilustra a situação e simula o funcionamento dos ponteiros do relógio.

```
my2dLoadIdentity();
my2dTranslate(c[0], c[1]); // x0=c[0], y0=c[1];
my2dRotate(-a);
my2dTranslate(-c[0], -c[1]);
transDrawClock(c, h);
```

No código acima, a matriz corrente na pilha de matrizes é carregada com a sua matriz identidade, depois ela é multiplicada pela matriz de translação $T(x_0, y_0)$, depois é multiplicada por uma matriz de rotação $R(-\theta)$, e finalmente, ela é multiplicada por uma de translação $T(-x_0, -y_0)$. Escrito em uma expressão, isto resulta em $[[[I]T(x_0, y_0)]R(-\theta)]T(-x_0, -y_0)$. Em ***transDrawClock()***, o centro do relógio ***c*** e final ***h*** são ambos transformados pela matriz corrente, e depois *scan* convertidos para o ecrã. Em OpenGL, a transformação está implícita, ou seja, os vértices são inicialmente convertidos pelo sistema antes de serem enviados para a conversão *scan*. O programa ***J2_1_Clock2d.java*** ilustra toda essa situação.

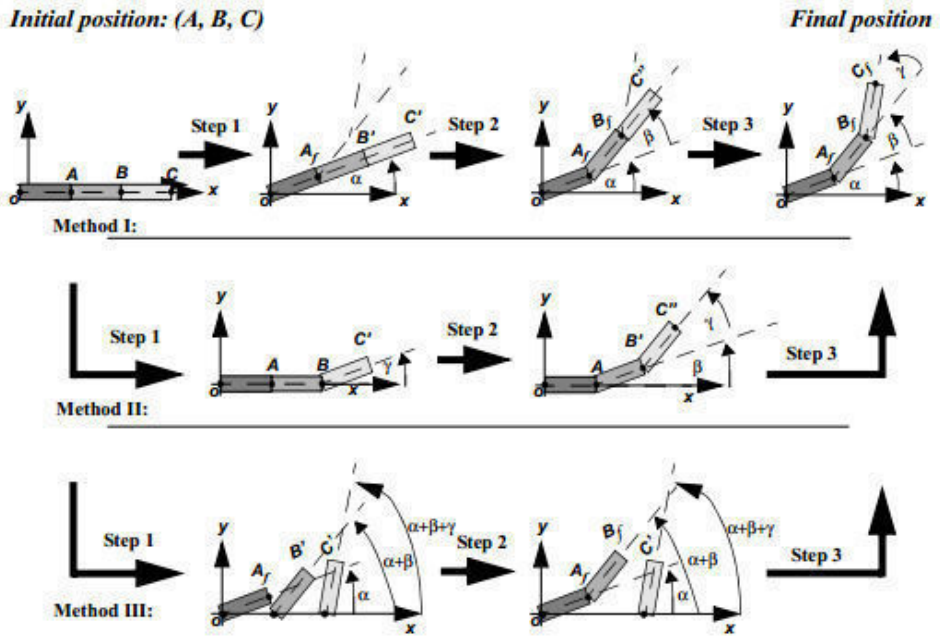
Exemplo 2: Redimensionamento de uma área rectangular. No OpenGL, é possível redimensionar uma área com o rato. Na função *callback* ***Reshape***, podemos utilizar ***glViewport()*** para ajustar o tamanho da área de desenho conforme o apropriado. O sistema executa os ajustes necessários através da mesma matriz de transformação. É a designada transformação janela-visor que será posteriormente discutida. Neste exemplo, a mesma situação é abordada, porém sem a utilização de ***glViewport()*** e sim, pela multiplicação de várias matrizes de transformação. Depois da transformação, a área rectangular bem como os seus vértices passam pela seguinte sequência de

transformações: translação de forma ao canto inferior esquerdo da área rectangular se posicionar na origem do sistema de coordenadas, alterar a escala de forma a ter a dimensão desejada, transladar de volta a posição original do rectângulo. A imagem abaixo, ilustra a situação. Em termos matriciais temos algo como: $T(P_2)S(s_x, s_y)T(-P_1)$



P_1 é a posição inicial para a alteração de escala, e P_2 é o destino. Podemos utilizar o rato para interactivamente arrastar P_1 para P_2 de forma a redimensionar a área rectangular correspondente. No código exemplo ***J2_2_Reshape.java***, é possível arrastar o vértice esquerdo mais abaixo P_1 para uma nova localização na área rectangular. O rectângulo e relógio no interior são redimensionados convenientemente.

Exemplo 3: Desenhar um braço 2D de um robot, incluindo o movimento dos seus segmentos. O braço 2D de um robot possui 3 segmentos que rodam nas suas articulações. Ele está localizado sobre um plano 2D. Considerada uma postura inicial *(A, B, C),* vamos encontrar as expressões matriciais para a transformação que resultem numa nova postura (A_f, B_f, C_f) com as respectivas rotações (α, β, γ) em torno das articulações. As localizações *(A, B, C)* são especificadas neste exemplo como estando todas situadas sobre o eixo x (e portanto, possuem y a 0) para facilitar a visualização e compreensão. De facto, *(A, B, C)* podem ser inicializados de forma arbitrária. Existem diversos métodos para alcançar o mesmo objectivo. A figura abaixo ilustra a situação e a forma como cada um dos métodos a aborda.



Método I

1. Girar $oABC$ em torno da origem por α graus:

$$A_f = R(\alpha)A;\ B' = R(\alpha)B;\ C' = R(\alpha)C$$

2. Considerar $A_fB'C'$ como um ponteiro de relógio, como no exemplo anterior. Girar $A_fB'C'$ em torno de A_f β graus. Isto é obtido primeiro movendo o ponteiro para a origem, girando e o devolvendo a posição original:

$$Bf = T(Af)R(\beta)T(-Af)B';\ C'' = T(A_f)R(\beta)T(-A_f)C'$$

3. Considerar o segmento B_fC'' também como um ponteiro de relógio. Girar B_fC'' em torno de B_f γ graus:

$$Cf = T(Bf)R(\gamma)T(-Bf)C''$$

No código ***J2_3_Robot2d.java*** está este e os demais métodos ilustrados. O método ***my2dTransHomoVertex(v1, v2)*** é responsável pela multiplicação da matriz corrente na pilha de matrizes com ***v1***, e gravação de resultados em ***v2***. ***drawArm()*** é utilizado para desenhar apenas o segmento de recta.

Método II

1. Considere BC ser um ponteiro de relógio. Gire BC em torno de B γ graus:

$$C' = T(B)R(\gamma)T(-B)C$$

2. Considere ABC' ser um ponteiro de relógio. Gire ABC' em torno de A β graus:

$$B' = T(A)R(\beta)T(-A)B;\ C'' = T(A)R(\beta)T(-A)C'$$

3. Novamente, considere $oAB'C''$ ser um ponteiro de relógio. Gire $oAB'C''$ em torno da origem α graus:

$$Af = R(\alpha)A;$$

$$Bf = R(\alpha)B' = R(\alpha)T(A)R(\beta)T(-A)B;$$

$$Cf = R(\alpha)C'' = R(\alpha)T(A)R(\beta)T(-A)T(B)R(\gamma)T(-B)C.$$

O método ***transDraw()*** transforma inicialmente os vértices, e depois desenha-os como um segmento de recta.

Método III

1. Considere oA, AB, e BC como sendo ponteiros de relógio com eixos de em o, A, e B, respectivamente. Gire oA α graus, AB (α+β) graus, e BC (α+β+γ) graus:

$$A_f = R(\alpha)A;\ B' = T(A)R(\alpha+\beta)T(-A)B;\ C' = T(B)R(\alpha+\beta+\gamma)T(-B)C$$

2. Mova AB' para A_fB_f:

$$B_f = T(A_f)T(-A)B' = T(A_f)R(\alpha+\beta)T(-A)B$$

Note que $T(-A)T(A) = I$, que é a matriz identidade: $I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. Qualquer matriz multiplicada pela sua identidade não se altera. O vértice movido pelo vector A, e depois devolvido a sua posição pelo vector inverso $-A$.

3. Mova BC' para B_fC_f:

$$C_f = T(B_f)T(-B)C' = T(B_f)R(\alpha+\beta+\gamma)T(-B)C$$

Translação, Rotação e Alteração de Escala em 3D

Em 3D, a translação ou a alteração de escala podem ocorrer no eixo z, além dos já usuais x e y. No caso da rotação, ela pode também ter uma componente em z (ângulo de rotação). Como no caso 2D, são utilizadas matrizes para calcular a transformação e todo o raciocínio utilizado anteriormente, é também neste caso válido (ver base teórica da UC).

Transformação Espacial no OpenGL

Até agora, executamos as transformações sem lançar mão das potencialidades disponíveis no OpenGL para este efeito. Para ilustrar, abordaremos novamente a implementação do braço de robot, só que com os métodos oferecidos pelo OpenGL. Vamos considerar um caso especial em 3D, em que z = 0.

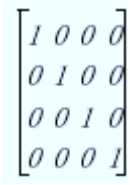
No OpenGL, todos os vértices de um modelo são multiplicados pela matriz no topo da pilha de matrizes MODELVIEW e em seguida pela matriz no topo da pilha de matrizes PROJECTION antes de ocorrer a conversão scan. As multiplicações são executadas no topo da pilha de forma automática pelo sistema gráfico. A pilha de matrizes MODELVIEW é utilizada para transformações geométricas. A PROJECTION é utilizada para visualização, sendo esta discutida mais tarde. Em seguida, é explicado como o OpenGL gere as transformações geométricas, considerando o exemplo de código ***J2_4_Robot.java*** (implementa o método II).

1. Especifica que as multiplicações correntes de matrizes devem ser executadas no topo da pilha MODELVIEW:

```
gl.glMatrixMode (GL.GL_MODELVIEW);
```

2. Carrega a matriz identidade na pilha:

```
gl.glLoadIdentity ();
```

A matriz identidade para coordenadas homogéneas em 3D é dada por: 

3. Especifica a matriz de rotação *R(α),* que será multiplicada pela matriz que esteja no topo da pilha actualmente. O resultado substitui a matriz que está no topo. Se a que está no topo for a identidade, então $IR_z(\alpha)=R_z(\alpha)$:

```
gl.glRotatef (alpha, 0.0, 0.0, 1.0);
```

4. Desenhar o braço do robot — um segmento de recta entre o ponto *O* e *A*. Antes do modelo ser convertido por *scan* para o *frame buffer*, *O* e *A* serão primeiramente transformados pela matriz no topo da pilha MODELVIEW, ou seja, por $R_z(\alpha)$. Isto significa que serão executadas as seguintes operações: $R_z(\alpha)O$ e $R_z(\alpha)A$:

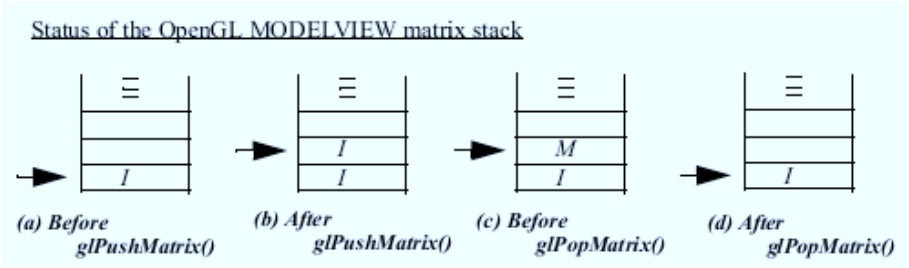
```
drawArm (O, A);
```

5. Na seção seguinte de código, é especificada uma série de matrizes de transformação, que por sua vez, serão multiplicadas pela matriz que esteja no topo da pilha: I, *[I]R(α),[[I]R(α)]T(A)*, *[[[I]R(α)]T(A)]R(β), [[[[I]R(α)]T(A)]R(β)]T(−A)*. Antes de ***drawArm(A, B)***, temos que $M = R(\alpha)T(A)R(\beta)T(-A)$ na pilha:

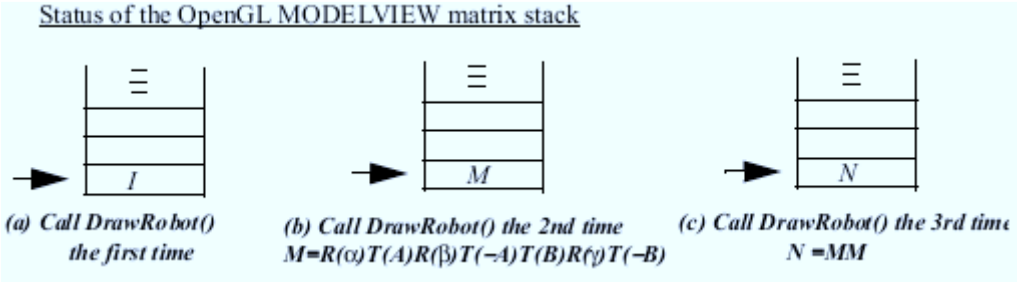
```
gl.glPushMatrix();
gl.glLoadIdentity ();
gl.glRotatef (alpha, 0.0, 0.0, 1.0);
drawArm (O, A);
gl.glTranslatef (A[0], A[1], 0.0);
gl.glRotatef (beta, 0.0, 0.0, 1.0);
gl.glTranslatef (-A[0], -A[1], 0.0);
drawArm (A, B);
gl.glPopMatrix();
```

A multiplicação matricial é sempre executada no topo da pilha. *glPushMatrix()* moverá o ponteiro da pilha para cima uma posição criando uma cópia da matriz que ali estava e colocando-a na nova posição. Dessa forma, fica-se com a mesma matriz nas duas posições (a de cima e a imediatamente abaixo). *glPopMatrix()* move o ponteiro da pilha uma posição abaixo. A vantagem deste mecanismo é a separação a transformação do modelo corrente entre *glPushMatrix()* e *glPopMatrix()* das outras transformações executadas no modelo mais tarde.

Vamos examinar melhor o método ***drawRobot()*** em ***J2_4_Robot.java***. A figura abaixo mostra o que está no topo da pilha, quando ***drawRobot()*** é chamado uma vez e outra em seguida. Em ***drawArm(B, C)*** antes da execução de um ***glPopMatrix(),*** a matriz no topo da pilha é $M = R(\alpha)T(A)R(\beta)T(-A)T(B)R(\gamma)T(-B)$.



6. Suponhamos que ***glPushMatrix()*** e ***glPopMatrix()*** são removidas de ***drawRobot(),*** se invocarmos ***drawRobot()*** uma vez, vai parecer que tudo está bem. Entretanto, se o invocarmos novamente, veremos que a matriz no topo da pilha não é uma matriz identidade. É novamente a matriz que lá estava antes. Veja a figura abaixo.



Os métodos I e III não podem efectivados utilizando-se directamente as transformações em OpenGL, pois o OpenGL permite a multiplicação entre matrizes, mas não entre vértices após estes terem sido transformados pela matriz transformação. Isso significa que todos os vértices estão sempre fixos em suas posições originais. Esta aproximação permite evitar o acúmulo de erros por sucessivos arredondamentos de casas decimais. Pode-se utilizar a função ***glGetDoublev(GL.GL_MODELVIEW_MATRIX, M[])*** para se obter os 16 valores correntes presentes na matriz no topo da pilha MODELVIEW e multiplicar as coordenadas pela matriz actual para se obter as transformações propostas nos métodos I e III.

Computação Gráfica (componente prática) - SEMANA 4

Temática: ***Eliminação de superfícies escondidas, desenho por subdivisão, composição de transformações 3D e detecção de colisões***

Actividade 4: Utilização do *zbuffer* para eliminação de superfícies ocultas, desenho de objectos 3D, composição de transformações em 3D e detecção de colisões com o JOGL

Competências a desenvolver:

- Aprender a activar o *zbuffer* em código para aumentar a performance gráfica
- Criar objectos geométricos básicos em 3D por subdivisão
- Executar composição de transformações em 3D
- Aprender a detectar colisões

Eliminação de Superfícies Ocultas

O algoritmo z-buffer (depth-buffer)

Em OpenGL, para activar o mecanismo de eliminação automática das superfícies ocultas de um modelo, é necessário permitir que seja executado o teste de profundidade aos vértices do mesmo e depois mandar limpar o *depth buffer* sempre que redesenhar uma *frame*:

```
// activa o zbuffer (depthbuffer) 1x
gl.glEnable(GL.GL_DEPTH_TEST);
// limpa o frame buffer e o zbuffer
gl.glClear(GL.GL_COLOR_BUFFER_BIT|GL.GL_DEPTH_BUFFER_BIT);
```

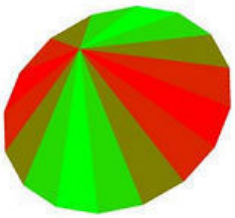
Correspondendo a um *frame buffer*, o sistema gráfico possui também um *z-buffer*, ou *depth buffer*, com o mesmo número de entradas. Depois de um ***glClear()***, o *z-buffer* é inicializado com o valor de ***z*** mais elevado em relação ao ponto de vista da cena, e o *frame buffer* é inicializado com a cor de fundo da cena. Quando é executada uma conversão *scan* do modelo (como p. ex. um polígono), antes de escrever um pixel colorido no *frame buffer*, o sistema gráfico (com base no algoritmo *z-buffer*) compara o valor de ***z*** do pixel ao ***z*** de seu correspondente (com mesmo valor de ***x*** e de ***y***) no *z-buffer*. Se o pixel estiver mais ponto em relação ao ponto de vista actual da cena, o seu valor ***z*** é escrito no *z-buffer* e sua cor no *frame buffer*. Caso contrário, o sistema segue para o próximo *pixel* sem escrever o actual nos buffers. O resultado é que a cena final contém apenas os *pixels* que não ocultam outros, não importante a ordem que a cena seja analisada.

Modelos 3D: Cone, Cilindro e Esfera

Aproximação de um cone. No exemplo de código em ***J1_5_Circle.java***, foi desenhado um círculo através da subdivisão de triângulos. Se o centro do círculo for elevado ao longo do eixo z, é possível se criar um cone. A figura abaixo ilustra a situação enquanto

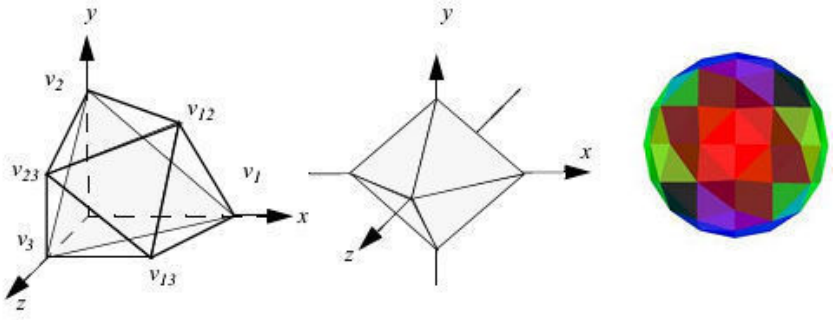
o código ***J2_5_Cone.java*** implementa a aproximação. Como o modelo está em 3D, é necessário activar a eliminação de superfícies ocultas. Outro pormenor, é garantir que o modelo está dentro dos limites do volume de visualização (ou seja, as coordenadas limites definidas):

```
gl.glOrtho(-w/2, w/2, -h/2, h/2, -w, w);
```



Aproximação de um cilindro. Se pudermos desenhar um círculo em $z = 0$, e depois outro em $z = 1$.Se conectarmos os rectângulos formados pelos mesmos vértices (isto, com mesmo x e y e diferente z) localizados nas arestas dos dois círculos, teremos um cilindro. Ver o código exemplo ***J2_6_Cylinder.java***.

Aproximação de uma esfera. Vamos assumir que temos um triângulo equilátero com três vértices (v_1, v_2, v_3) sobre a esfera *e* $|v_1|=|v_2|=|v_3|=1$. Isto é, os três vértices são vectores unitários a partir da origem. Podemos constatar que $v_{12} = normalize(v_1 + v_2)$ está também situado sobre a esfera. Podemos subdividir o triângulo em quadro triângulos equiláteros como a figura abaixo ilustra. O exemplo implementado em código pode ser visto em ***J2_7_Sphere.java***. É utilizado este método para subdividir um octaedro.

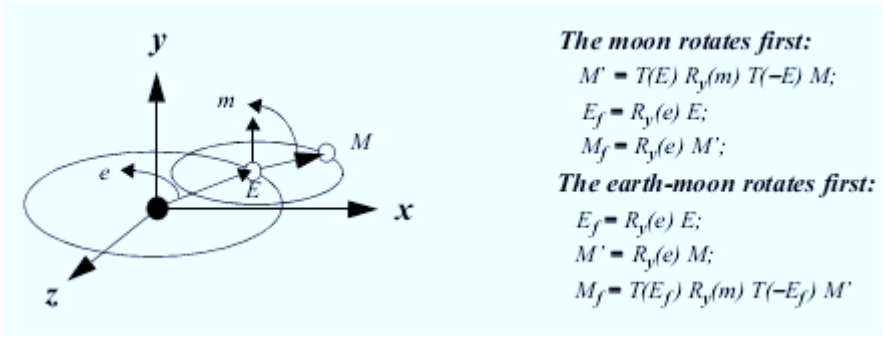


(a) Subdivision *(b) Front view of an octahedron* *(c) A sphere*

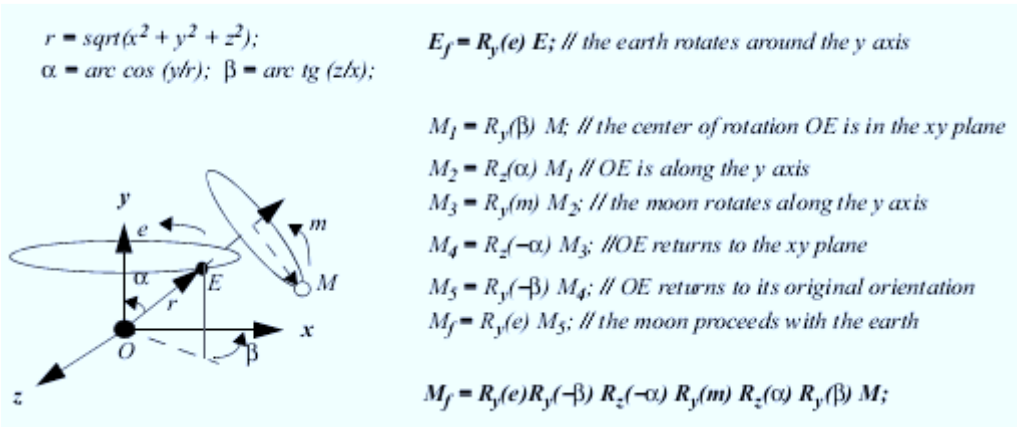
Composição de transformações 3D

O exemplo ***J2_8_Robot3d.java*** implementa o braço de robot no exemplo ***J2_4_Robot.java*** com cilindros 3D. Também foi adicionada uma rotação em torno de y de forma ao robot girar o braço em 3D.

Um outro bom exemplo de composição de transformações em 3D é dado pelo programa ***J2_9_Solar.java***. Ele implementa um sistema solar simplificado. A Terra gira em torno do Sol e a Lua, em torno da Terra no plano xz. Considerando-se o centro da Terra em $E(x_e, y_e, z_e)$ e centro da Lua em $M(x_m, y_m, z_m)$, vamos calcular os novos centros depois que a Terra gira em torno do Sol e graus, e a Lua gira em torno da Terra m graus. A Lua também gira em torno do Sol com a Terra. A figura abaixo ilustra a situação.



O problema é similar ao do relógio com excepção do centro do relógio estar também a girar em torno do eixo *y*. Podemos considerar a Lua a girar em torno da Terra primeiramente, e depois a Lua e a Terra como um único objecto a girar em torno do Sol. No OpenGL, como podemos desenhar uma esfera no centro do sistema de coordenadas, a transformação torna-se mais simples.



Em seguida, vamos alterar o sistema solar para um sistema mais complexo, que será designado de sistema solar generalizado. Nesse, a Terra é elevada ao longo do eixo do *y*, e a Lua é elevada ao longo de um eixo que vai da origem do sistema de coordenadas na direcção do centro da Terra (eixo arbitrário), e a Lua gira em torno desse eixo como mostrado na figura abaixo. Dito de outra forma, a Lua gira em torno do vector *E*. Dados *E* e *M* e seus respectivos ângulos de rotação *e* e *m*, podemos encontrar as novas coordenadas de E_f e M_f?

Não é possível obter de forma imediata a matriz de rotação *M* para a Lua. Entretanto, podemos considerar *E* e *M* como um único objecto e criar a matriz de rotação através de várias etapas. Note que para girar *M* em torno de *E*, não é realmente necessário girar *E*, porém a utilizamos como uma referência para explicar a rotação.

1. O ângulo entre o eixo y e *E* é $\alpha = arc\ cos\ (y/r);$ o ângulo entre a projecção de *E* no plano *xz* e eixo do *x* é $\beta = arc\ tg\ (z/x);\ r = sqrt(x_2 + y_2 + z_2)$.

2. Girar *M* em torno de y de β graus de forma ao novo centro de rotação E_1 ficar no plano *xy*:
$$M_1 = Ry(\beta)M;\ E_1 = Ry(\beta)E.$$

3. Girar M_1 em torno do eixo *z* de α graus de forma ao novo centro de rotação E_2 ficar coincidente com o eixo do *y*:

$$M_2 = R_z(\alpha)M_1;\ E_2 = R_z(\alpha)E_1$$

4. Girar M_2 em torno do eixo *y* *m* graus:

$$M_3 = R_y(m)M_2$$

5. Girar M_3 em torno do eixo *z* $-\alpha$ graus de forma ao centro da rotação retorne ao plano *xz*:

$$M_4 = R_z(-\alpha)M_3;\ E_1 = R_z(-\alpha)E_2$$

6. Girar M_4 em torno do eixo *y* $-\beta$ graus de forma ao centro da rotação retorne a sua orientação original:

$$M_5 = R_y(-\beta)M_4;\ E = R_y(-\beta)E_1$$

7. Girar M_5 em torno do eixo *y* *e* graus de forma a Lua avançar com a Terra em torno do eixo *y*:

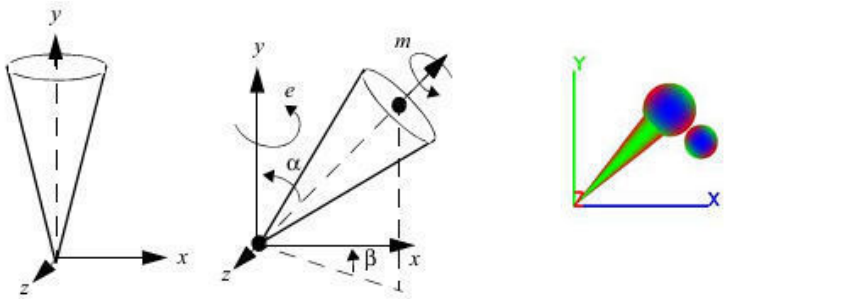
$$M_f = R_y(e)M_5;\ E_f = R_y(e)E$$

8. Colocando todas as matrizes de transformação juntas, temos:

$$M_f = R_y(e)R_y(-\beta)\ R_z(-\alpha)\ R_y(m)\ R_z(\alpha)\ R_y(\beta)\ M$$

Novamente, em OpenGL, inicia-se com uma esfera no centro da origem. A transformação é bem mais simples. O código disponível no programa ***J2_10_GenSolar.java*** implementa um sistema solar generalizado. Incidentalmente, ***glRotatef(m, x, y, z)*** especifica uma única matriz que gira um ponto ao longo do *vector(x, y, z) m* graus. Ou seja, a matriz é igual a $R_y(-\beta)\ R_z(-\alpha)\ R_y(m)R_z(\alpha)\ R_y(\beta)$.

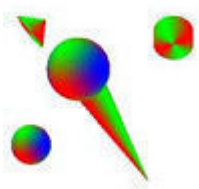
O sistema solar generalizado corresponde a uma espécie de cone que gira e avança conforme a figura abaixo ilustra. O ângulo de rotação é *m* e o de avanço é *e*. A Terra *E* está num ponto ao longo do centro do cone, e a Lua *M* pode ser um ponto na extremidade desse cone. Sabemos como desenhar um cone em OpenGL. Podemos executar transformações no cone para saber o movimento do mesmo. O código apresentado em ***J2_11_ConeSolar.java*** ilustra exactamente essa situação – um cone que gira e avança e uma esfera que gira em torno de sua base.



(a) A top *(b) Rotating and proceeding* *(c) A Top in generalized solar system*

Detecção de Colisões

Para evitar que dois modelos durante uma animação penetrem um no outro, pode-se considerar volumes envoltórios externos, que definem limites, que por sua vez permitam avaliar se há ou não colisão. Os volumes envoltórios podem ter formas variadas, como caixas ou esferas. O exemplo anterior pode ser alterado de forma a termos 3 Luas (um cilindro, uma esfera e um cone) que giram em torno da Terra com direcções diferentes e colidem um com o outro, mudando a direcção de rotação. Se utilizarmos uma esfera como volume envoltório limite, o problema será encontrar os centros das esferas envolventes. Sabemos que cada Lua é transformada a partir de sua origem. Se conhecermos a matriz corrente na pilha de matrizes no momento em que se desenha a Lua, podemos multiplicar essa matriz pela origem *(0, 0, 0, 1)* para encontrar o centro da Lua. Como a origem x, y e z estão a zero, só é necessário recuperar a última coluna da matriz. O exemplo ***J2_11_coneSolarCollision.java*** implementa e ilustra esta situação. A colisão é decidida em função da distância entre os centros das Luas. Se a distância for menor do que um valor limite pré-definido, as duas Luas mudam de direcção de rotação em torno da Terra. A figura a baixo ilustra o resultado do programa.



Computação Gráfica (componente prática) - Parte 5

Temática: ***Pipeline de visualização em 2D e 3D***

Actividade 5: Aplicação do pipeline de visualização com o JOGL

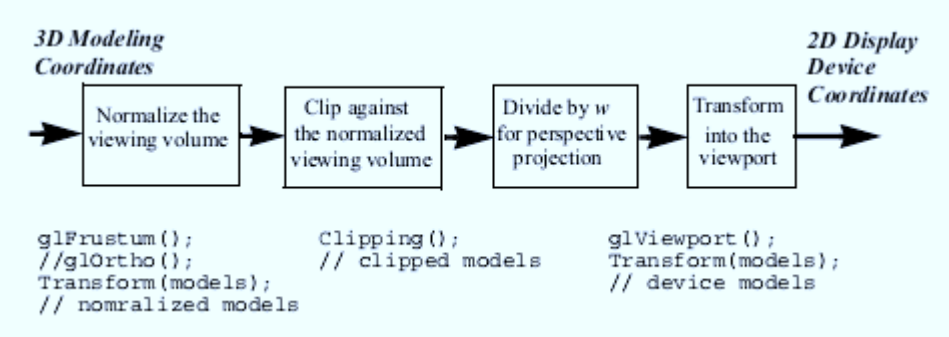
Competências a desenvolver:

- Conhecer a visualização 2D e 3D
- Aprender a ordem lógica e passos para a transformação do *viewing pipeline* com o JOGL
- Conhecer a biblioteca GLU
- Saber criar vários visores (*viewports*)

Visualização

O dispositivo de visualização possui as suas coordenadas em *pixels*, enquanto o modelo a ser desenhado (virtual) possui o seu próprio sistema de coordenadas (metros, centímetros, etc.). É necessário estabelecer uma correlação entre as coordenadas do dipositivo de visualização e as do modelo, de forma a ser possível visualiza-lo no ecrã. Logo, é preciso executarem-se transformações visuais – mapear uma área ou volume do modelo (em coordenadas do modelo) para uma área no dispositivo de visualização (em coordenadas do dispositivo).

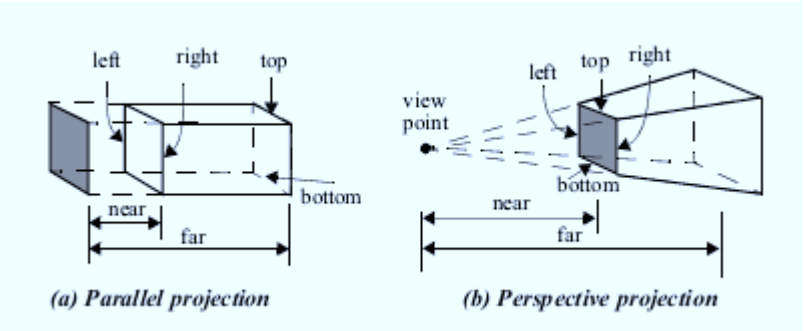
No OpenGL (e logo também no JOGL) é executado o designado *viewing pipeline*, que inclui etapas de normalização, recorte, divisão perspectiva e transformação janela-visor (ver figura abaixo). Com excepção do recorte, todas as demais transformações podem ser obtidas através da multiplicação de matrizes. Isto é, a visualização é na sua maioria obtida através de transformações geométricas. No OpenGL isto é alcançado com a multiplicação das matrizes que estão na pilha de matrizes ***PROJECTION***.



Especificação de um volume de visualização

Uma projecção paralela (ver figura abaixo) é designada de ortogonal se todas as projectantes são perpendiculares ao plano de visualização. ***glOrtho(left, right, bottom, top, near, far)*** especifica a projecção ortográfica. ***glOrtho()*** também define as equações dos seis planos que formam o volume de visualização (paralelepípedo): $x = esquerda$, $x = direita$, $y = abaixo$, $y = acima$, $z = -perto$, e $z = -longe$. Nós podemos ver que

(esquerda, abaixo, -perto) e *(direita, acima, -perto)* especificam as coordenadas *(x, y, z)* dos cantos inferior esquerdo e superior direita do plano de recorte mais próximo. De forma similar, *(esquerda, abaixo, -longe)* e *(direita, acima, -longe)* especificam as coordenadas *(x, y, z)* dos cantos inferior esquerdo e superior direito do plano de recorte mais distante.



glFrustum(left, right, bottom, top, near, far) especifica uma projecção em perspectiva (ver figura acima). ***glFrustum()*** também define seis planos que cobrem o volume de visualização em perspectiva (pirâmide). Podemos ver que *(esquerda, abaixo, -perto)* e *(direita, acima, -perto)* especificam as coordenadas *(x, y, z)* dos cantos inferior esquerdo e superior direito do plano de projecção mais próximo. O plano de recorte mais distante está em *z = -longe* com as projectantes convergindo para o ponto de vista, o qual está posicionado na origem e orientado na direcção negativa do eixo *z*.

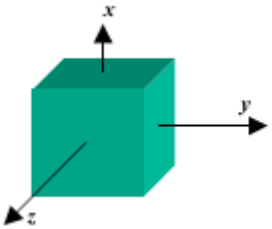
Como podemos ver, tanto ***glOrtho()*** como ***glFrustum()*** especificam volumes de visualização cujas arestas direita e esquerda do plano mais próximo, estão paralelas ao eixo do *y*. Em geral, é utilizado o ***vector up*** para representar a orientação do volume de visualização. Este vector quando projectado no plano mais próximo fica paralelo as arestas esquerda e direita desse plano.

Normalização

É a transformação obtida pela multiplicação de matrizes presentes na pilha de matrizes ***PROJECTION***. No código abaixo, é inicialmente carregada a matriz identidade no topo da pilha. Em seguida, ela é multiplicada pela matriz especificada com o auxílio de ***glOrtho().***

```
// É activado o modo de projecção e respectiva pilha
gl.glMatrixMode (GL.GL_PROJECTION);
gl.glLoadIdentity ();
gl.glOrtho(-Width/2,Width/2,-Height/2,Height/2,-1.0, 1.0);
```

No OpenGL, ***glOrtho()*** especifica a matriz que transforma o volume de visualização num volume normalizado de visualização, ou seja, num cubo com seis planos de recorte (ver figura abaixo)localizados em ($x = 1, x = -1, y = 1, y = -1, z = 1,$ e $z = -1$). Em lugar de se calcular directamente o recorte e a projecção, é executada a normalização primeiramente, simplificando as etapas seguintes a nível de cálculos.



De forma idêntica, ***glFrustum()*** também especifica uma matriz que transforma o volume de visualização num volume normalizado em perspectiva. Neste caso, é necessária uma divisão para mapear as coordenadas homogéneas em coordenadas 3D. No OpenGL o vértice é representado por *(x, y, z, w)* e as matrizes de transformação por matrizes. Quando $w = 1$, *(x, y, z)* representa as coordenadas 3D do vértice. Se $w = 0$, *(x, y, z)* representa a direcção. Em caso contrário, *(x/w, y/w, z/w)* representa as coordenadas 3D. A divisão perspectiva é necessária porque depois da transformação matricial causada por ***glFrustum()***, $w \neq 1$. No OpenGL, a divisão perspectiva é executada depois do recorte.

Recorte

Como ***glOrtho()*** e ***glFrustum()*** transformam os seus volumes de visualização num volume normalizado, só é necessário executar um algoritmo de recorte. O recorte é efectuado em coordenadas homogéneas para acomodar certas curvas. Por isso, todos os vértices do modelo são inicialmente normalizados, recortados em relação aos planos normalizados do volume de visualização ($x = -w$, $x = w$, $y = -w$, $y = w$, $z = -w$, $z = w$), e depois transformados e projectado no visor 2D (janela de visualização no ecrã).

Divisão em perspectiva

A normalização causada por ***glFrustum()*** resulta em coordenadas homogéneas com $w \neq 1$. O recorte é executado em coordenadas homogéneas. Entretanto, uma divisão em todas as coordenadas do modelo *(x/w, y/w, z/w)* é necessária para transformar as coordenadas em coordenadas 3D.

Transformação janela-visor

Todos os vértices são mantidos em 3D. São necessários os valores de z para a eliminação das superfícies ocultas. A partir do volume normalizado e divido por w, a transformação janela-visor calcula cada vértice *(x, y, z)* correspondente aos *pixels* presentes no visor e invoca os algoritmos de conversão *scan* para desenhar o modelo na janela. A projecção em 2D ignora o valor de z quando é executada a conversão *scan* para o *frame buffer*. Não é necessário, mas podemos considerar que a projecção ocorre sobre um plano em $z = 0$.

Resumo da *viewing pipeline* no OpenGL:

- **Modelação:** cada vértice do modelo é transformado pela matriz actual que está no topo da pilha de matrizes ***MODELVIEW***.
- **Normalização:** Depois da etapa anterior, cada vértice será transformado pela matriz actual que está no topo da pilha de matrizes ***PROJECTION***.
- **Recorte:** Cada primitiva (pontos, polígonos, etc.) é recortada contra os planos de recorte que estão em coordenadas homogéneas.
- **Divisão em perspectiva:** Todas as primitivas são transformadas de coordenadas homogéneas em coordenadas cartesianas.
- **Transformação janela-visor:** O modelo é escalonado e transladado no visor para conversão scan.

A ordem lógica das etapas de transformação

O exemplo ***J2_12_RobotSolar.java*** demonstra como especificar as matrizes de projecção e modelo (***PROJECTION*** e ***MODELVIEW***) nas duas respectivas pilhas de matrizes. A ideia é saber onde se especifica o modelo e para onde se deve especificar a matriz de projecção.

1. No método ***display(),*** é calculada a origem do braço do robot à partir das coordenadas de modelação.
2. Como referido antes, embora as matrizes sejam multiplicadas de cima para baixo com o auxílio dos comandos de transformação, quando analisamos as transformações no modelo, em termos lógicos, a etapas de transformação são de baixo para cima, ou seja, da primeira transformação especificada para o modelo, presente na pilha ***MODELVIEW*** até a especificação do volume de visualização na matriz presente na pilha ***PROJECTION***.
3. O OpenGL oferece estas duas pilhas de matrizes para facilitar a gestão lógica da visualização e transformação de forma separada. Em termos teóricos, não é necessária a existência de tal separação, logo, podemos considerar essas duas matrizes como resultando numa única expressão matemática. Veja o exemplo em 4.
4. Em ***Reshape()***, o braço do robot é movido ao longo do eixo do z de $-(zNear + zFar)/2$ de forma a ser posicionado no meio do volume de visualização. Essa transformação pode estar definida na primeira matriz em ***MODELVIEW*** ou na última em ***PROJECTION***.
5. ***glOrtho()*** ou ***glFrustrum()*** especificam um volume de visualização. Os modelos no volume de visualização aparecerão na janela-visor.
6. ***glViewport()*** em ***Reshape()*** especifica uma área de desenho dentro da janela de visualização. O volume de visualização será projectado na área correspondente do visor. Quando redimensionamos a área de desenho, por exemplo, a proporção do visor (w/h) altera-se convenientemente. Podemos especificar outro visor dentro da mesma janela com ***glViewport(),*** ou seja, podemos ter múltiplos visores numa mesma janela.

Uma outra forma de olhar para a transformação do modelo e de visualização é a de considerar que a expressão matricial transforma a forma de visualização (ou forma de olhar para o modelo) em lugar do modelo. Mover o modelo ao longo do eixo z negativo é

equivalente a mover o volume de visualização (camara) ao longo do eixo *z* positivo. De forma similar, girar o modelo em torno de um eixo com um ângulo positivo equivale a girar o volume de visualização em torno do mesmo eixo e ângulo, porém com valor negativo. Quando analisamos a transformação de um modelo segundo as transformações de sua visualização, a ordem de transformações é de cima para baixo, ou seja, iniciamos com o volume de visualização e terminamos com os comandos de desenho. Outro detalhe a ter em conta é que a sinalização da transformação é logicamente negativa nesta perspectiva. O exemplo ***J2_12_RobotSolar.java***, ilustra em ***myCamera()*** uma transformação de cima para baixo.

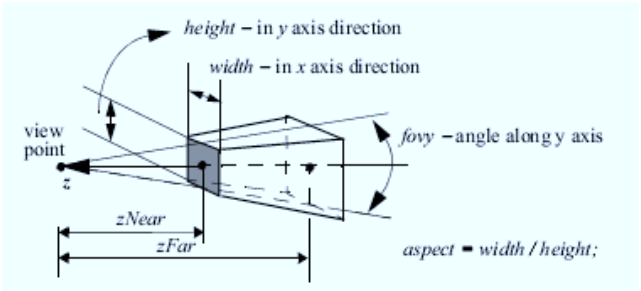
gluPerspective e gluLookAt

A biblioteca GLU (*OpenGL Utility*), que é considerada parte do OpenGL, possui várias funções de conveniência que foram construídas à partir de funções do OpenGL com o intuito de complementa-lo. O prefixo é *"glu"* em lugar de *"gl."* Para uma melhor compreensão da visualização, serão discutidas as funções da GLU: ***gluPerspective()*** e ***gluLookAt().*** Demais funções da GLU serão discutidas mais a frente nesta UC.

A ***gluPerspective()***configure a matriz de projecção da seguinte forma:

```
void gluPerspective(
        double fovy, // the field of view angle in y-direction
        double aspect, // width/height of the near clipping plane
        double zNear, // distance from the origin to the near
        double zFar // distance from the origin to far
);
```

Os parâmetros estão ilustrados na figura abaixo. Comparado com a ***glFrustum(), gluPerspective()*** é mais simples para os programadores, porém menos potente. O ângulo *fovy* (campo de vista) é simétrico em tono do eixo *z* na direcção *y*, e os planos perto e longe de recorte são também simétricos em torno do eixo do *z*. Por isso, ***gluPerspective()*** só consegue especificar um volume simétrico em torno de *z*, ao contrário de ***glFrustum()*** que não impõe tal restrição.



O exemplo ***J2_14_Perspective.java*** ilustra uma implementação com ***myPerspective(double fovy, double aspect, double near, double far). glOrtho(),glFrustum(),*** e ***gluPerspective*** especificam um volume de visualização com arestas direita e esquerda do plano perto de recorte paralelas ao eixo y. Como referido antes, o *vector up* representa a orientação do volume de visualização. Em outras palavras, por defeito, a projecção do *vector up* no plano de recorte é sempre paralela a *y*. Como podemos transformar o volume de visualização, se especificarmos a orientação do vector *(upX, upY, upZ),* podemos orientar o volume de visualização convenientemente. O ângulo

entre *y* e a projecção de *up* no plano *xy* é dada por *atan(upX/upY).* É necessário apenas girar o volume de visualização *–atan(upX/upY)* para obter isso. Isto ainda pode ir mais longe. Não é necessário olhar para baixo a partir da origem, na direcção negativa de *z*. Em lugar disso, podemos especificar o ponto de vista como um ponto de mirada para baixo, com foco num outro ponto central, com o *vector up* identificando a orientação do volume de visualização. Parece complexo, mas uma transformação equivalente é bem mais simples.
Considerando-se um triângulo em 3D (*eye, center, up*), podemos construir uma matriz de transformação que depois desta ocorrer, o *eye* esteja na origem, o *center* no eixo negativo de *z*, e o *up* no plano *yz*? O método ***myLookAt()*** no código ***J2_15_LookAt.java*** demonstra isso. ***myLookAt()*** e ***myGluLookAt()*** no exemplo são equivalentes a simulações de ***gluLookAt(),*** a qual define um volume de transformação a partir de *eye* para outro ponto identificado por *center* com *up* como o vector orientação do volume de visualização:

```
void gluLookAt (double eyeX
        , double eyeY
        , double eyeZ
        , double centerX
        , double centerY
        , double centerZ
        , double upX
        , double upY
        , double upZ
);
```

O *eye* e o *center* são pontos, enquanto *up* é um vector. Isto é ligeiramente diferente do exemplo com o triângulo, onde *up* é também um ponto. Como podemos constatar, o vector *up* não pode ser paralelo a recta definida por (*eye, center*).

Múltiplos visores

glViewport(int x, int y, int width, int height) especifica uma area dentro da janela de visualização. Por defeito, ***glViewport(0, 0, w, h***) é implicitamente invocada a cada ***reshape(GLDrawable glDrawable, int x, int y, int w, int h)*** com área idêntica a da janela de visualização. O volume de visualização é projectado convenientemente nessa área. Podemos entretanto especificar diferentes visores com ***glViewport()*** com valores para o canto inferior direito variando de *(0, 0)* a *(w, h).* Todas as funções de redesenho desenharão dentro dessa área definida pelo visor. Também podemos especificar múltiplos visores com áreas diferentes e desenhar cenas diferentes em cada um. Por exemplo, ***glViewport(0, 0, width/2, height/2)*** utiliza a quarta parte inferior esquerda da janela de visualização, e ***glViewport(width/2, height/2, width/2, height/2)*** utiliza a quarta parte superior a direita. No exemplo ***J2_15_LookAt.java***, são utilizados diferentes métodos de projecção para demonstrar as funções ***myLookAt(), mygluLookat()***, e ***myPerspective().*** Se não forem utilizados métodos diferentes para os diferentes visores, a mesma matriz de projecção será utilizada nos diferentes visores.

Computação Gráfica (componente prática) - Parte 6

Temática: ***Cor e luz nas cenas gráficas***

Actividade 6: Utilização da cor e da luz em cenas criadas com o JOGL

Competências a desenvolver:

- Aprender os modelos de cor utilizados no JOGL
- Aprender a utilizar as diversas componentes do material dos objetos para criar efeitos luminosos diferentes
- Aprender a ativar e criar fontes de luz em cenas gráficas com o JOGL

Modo RGB e Modo Índice

Cada valor de *pixel* no *frame buffer* corresponde a um vector RGB, pois o dispositivo de visualização opera no modo RGB. Cada valor de *pixel* também pode ser um índice numa tabela de cor do tipo *look-up*, designada de *colormap*. Neste caso, o dispositivo de visualização opera no modo índice de cor. A cor do *pixel* passa então a ser especificada pela valor da entrada na tabela *colormap* em vez do valor contido no *frame buffer*.

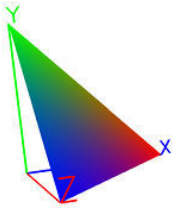
A tabela *colormap* não ocupa muito a memória. Sua utilização foi uma grande vantagem antes do aparecimento de chips de memória rápidos e baratos. Na GLUT é utilizada a função ***glutInitDisplayMode(GLUT_INDEX)*** para activar o modo índice de cor. O modo RGB é sempre assumido por defeito. O modo índice também pode ser útil para se executar alguns truques de animação. Entretanto, em geral, por causa da memória já não ser mais uma limitação e o modo RGB ser mais fácil e mais flexível, ele é utilizado nos exemplos mais a frente. Em OpenGL, cada componente de cor *(R, G, ou B)* pode variar com valores entre 0 a 1. O sistema adequa a cor final em função da capacidade da placa gráfica disponível, sendo este processo transparente para o utilizador final.

Interpolação da cor

No OpenGL, é utilizado a ***glShadeModel(GL_FLAT)*** ou ***glShadeModel(GL_SMOOTH)*** para optar entre os dois modos distintos de colorir modelos (*flat shading e smooth shading*). Com ***GL_FLAT***, é utilizada uma cor que é especificada com ***glColor3f()*** para todos os *pixels* da primitiva. No exemplo ***J3_1_Shading.java***, se a ***glShadeModel(GL_FLAT)*** é invocada, apenas uma cor será utilizada em ***drawtriangle(),*** mesmo que tenhamos especificado cores diferentes para cada um dos vértices. Dependendo do sistema OpenGL, a cor pode ser a que foi definida para o último vértice da primitiva.

Para um segmento de recta, com ***GL_SMOOTH***, as cores dos vértices são linearmente interpoladas ao longo dos *pixels* entre os dois vértices. Por exemplo, se um segmento tiver 5 *pixels*, e as extremidades tiverem cores *(0, 0, 0)* e *(0, 0, 1),* então, depois da interpolação, os 5 pixels terão as seguintes cores respectivamente: *(0, 0, 0), (0, 0, 1/4), (0, 0, 2/4), (0, 0, 3/4),* e *(0, 0, 1).* A intensidade de cada componente RGB é interpolada separadamente.

Para um polígono, o OpenGL interpola primeiro ao longo das arestas, e depois ao longo das linhas horizontais de *scan* durante a conversão *scan*. A única coisa necessária a fazer para que isto ocorra é invocar a função ***glShadeModel(GL_SMOOTH)*** e definir cores diferentes para os vértices como ilustrado na figura abaixo.



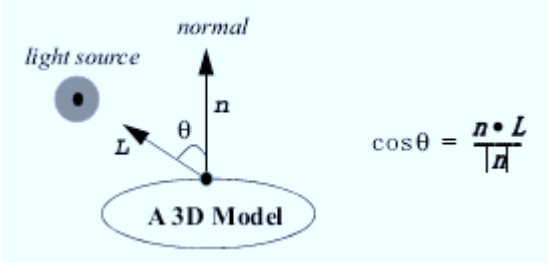
Iluminação

O sistema de iluminação do OpenGL inclui quatro componentes maiores: ambiente, difusa, especular e emissiva. A cor final é resultante do somatório dessas componentes. O modelo de iluminação é desenvolvido para calcular a cor em cada *pixel* individual que corresponda a um ponto na primitiva a ser desenhada. Esse método é designado de modelo de sombreamento. Como já referido, o OpenGL calcula a cor dos pixels do vértice e aplica interpolação para encontrar a cor para todos os demais pixels da primitiva quando a função ***glShadeModel(GL_SMOOTH)*** é invocada. Se utilizarmos ***glShadeModel(GL_FLAT)***, somente a cor de um dos vértices é considerada para a primitiva. Entretanto, as cores dos vértices são calculadas pelo modelo de iluminação em lugar do especificado por ***glColor().***

No exemplo ***J3_2_Emission.java***, o material emite uma cor branca e todos os objetos ficarão a branco até que alteremos a componente emissiva para algo diferente de branco. Se especificarmos apenas a componente emissiva (neste caso a branco), o efeito é o mesmo de especificar ***glColor3f(1., 1.,1.).***

Quando a iluminação está activada, ***glColor3f()*** é desactivado. Mesmo que tenhamos ***glColor3f()s*** no programa, eles não serão utilizados. Em lugar disso, o sistema OpenGL utilizará o modelo de iluminação actual para calcular a cor no vértice automaticamente. Podemos utilizar ***glColorMaterial()*** com ***glEnable(GL_COLOR_MATERIAL)*** para associar a cor especificada com ***glColor3f()*** a propriedade do material.

A cor ambiente é a intensidade geral de múltiplas reflexões geradas a partir de uma fonte luminosa num determinado ambiente. Nesse caso, não nos preocupamos com a localização da fonte luminosa, mas sim com a sua existência. No exemplo ***J3_3_Ambient.java***, é mostrada a forma como a cor é obtida levando em conta a cor ambiente (que neste caso, resulta num valor RGB igual a (1., 1., 0)).



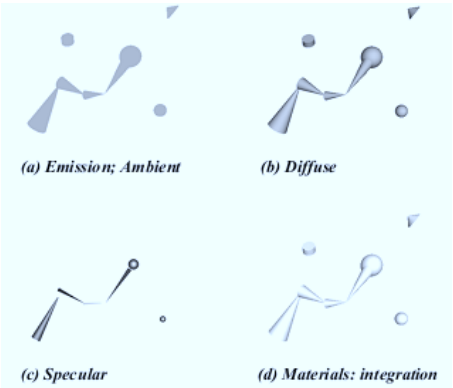
A cor difusa resulta de um material cuja superfície é igualmente brilhante em todas as direcções de visualização. No OpenGL, *L* é um vector unitário (ou normalizado) que aponta

para a fonte de luz a partir do vértice actual. A normal é especificada por ***glNormal*()*** antes de se especificar o vértice. Podemos especificar a normal para ser um vector unitário. Entretanto, as normais são transformadas de forma similar aos vértices de maneira aos seus comprimentos serem escalonados. Se não estivermos seguros em relação ao comprimento das normais, podemos invocar ***glEnable(GL_NORMALIZE),*** a qual activa a normalização das normais antes de calcular a iluminação. Isso causa cálculos extras. A fonte luminosa é composta por quatro parâmetros: *(x, y, z, w)* em coordenadas homogéneas. Se *w* é 1, *(x, y, z)* é a posição da fonte luminosa. Se *w* é 0, *(x, y, z)* indica a direcção da fonte luminosa situada no infinito (neste caso, a fonte luminosa está na mesma direcção para todos os pixels, localizados em diferentes localizações). Se uma fonte luminosa está longe de um objecto, seus raios farão essencialmente o mesmo ângulo com todas as superfícies que possuam normais com direcção idêntica. O exemplo ***J3_4_Diffuse.java*** ilustra a utilização dos parâmetros para luz difusa no OpenGL. Dependendo de como as normais são definidas, também o efeito final de luz e sombra varia muito. Por exemplo, consideremos uma superfície triangular. Se quisermos que o efeito visual pareça uma pirâmide as normais dos vértices de sua face devem ser iguais e perpendiculares ao triângulo. Se quisermos que pareça com um cone, as normais devem ser perpendiculares a superfície do cone. O sistema interpolará as cores dos *pixels* resultando em diferenças visuais.

A cor especular é a reflexão de uma superfície feita com material liso que depende da direcção de reflexão dada por *R* (que é *L* reflectido ao longo da normal) e da direcção de visualização *V*. O ponto de observação está na origem. Podemos utilizar ***glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE)*** para especificar o ponto de vista em *(0, 0, 0).* Entretanto, para simplificar os cálculos, OpenGL permite que o ponto de vista seja especificado numa posição infinita, na direcção *(0, 0, 1)* direcção. Este é o valor por defeito, sendo a direcção igual para todos os *pixels*. Como este valor de ponto de vista é utilizado para simplificar os cálculos, ele não é alterado para os demais cálculos, como o de projecção. O exemplo ***J3_5_Specular.java*** mostra como definir parâmetros especulares em OpenGL.

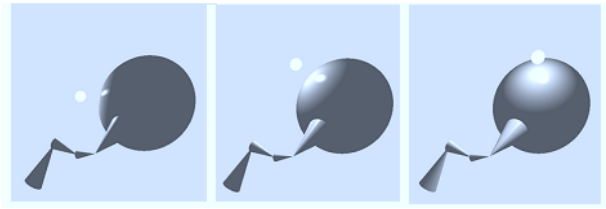
Modelo de iluminação no OpenGL

Tanto a fonte luminosa como o material possuem múltiplos componentes: ambiente, difuso e especular. A cor final do vértice é uma integração de todos esses componentes. Nos exemplos anteriores, mesmo sem especificar todos os componentes, o OpenGL aplica valores por defeito pré-definidos. Se necessário, podemos especificar componentes de luz diferentes (***Example J3_6_Materials.java***). A figura abaixo ilustra os resultados diferentes consoante a parametrização executada.



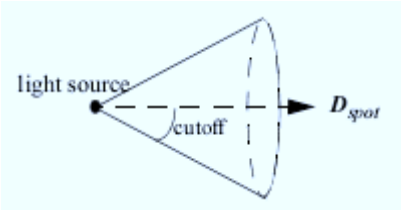
Fonte de luz móvel

No OpenGL, uma fonte de luz é invisível. A posição da fonte de luz é transformada num objecto geométrico pela matriz actual, quando esta é especificada. Em outras palavras, se a matriz for modificada em *runtime*, a fonte luminosa pode ser movida como qualquer outro objecto. A iluminação é calculada consoante a posição modificada. Para simular fontes de luz visíveis, podemos especificar a fonte de luz e desenhar um objecto (por exemplo uma lâmpada) na mesma posição. No exemplo ***J3_7_MoveLight.java***, a fonte de luz é uma esfera e tanto ela como a fonte de luz são transformadas pela mesma matriz. Podemos definir as propriedades emissivas da esfera de forma a corresponder aos parâmetros da fonte de luz,e assim, a esfera parecer uma fonte de luz (ver figura abaixo).



Efeito Spotlight

Uma fonte de luz real pode não gerar a mesma intensidade de luz em todas as direcções (ver figura abaixo). Neste caso, não há luz fora da zona cónica do foco. Para ser exacta, a área do cone é infinita na direcção *D* do foco de luz. O exemplo ***J3_8_SpotLight.java*** ilustra a especificação de parâmetros para fontes de luz do tipo *spot*. O vector D do foco de luz é também transformado pela matriz actual *modelview*, bem como as normais aos vértices.



Atenuação da fonte de luz

A intensidade da luz a partir da fonte até um vértice pode ser atenuada pela distância. O exemplo ***J3_9_AttLight.java*** mostra como especificar esses factores.

Múltiplas fonts e luz

Podemos também definir várias fontes de luz: Cada uma com os seus parâmetros e posições. Poderão existir fontes estáticas ou móveis. A componente emissiva, que é uma propriedade do material, não depende de nenhuma fonte luminosa. Podemos utilizar ***glLightModel()*** para definir a luz ambiente global que não depende de nenhuma outra fonte de luz. ***J3_10_Lights.java*** ilustra essa situação.

Sombreamento de superfícies visíveis

Os modelos de sombra são métodos para calcular a iluminação nas superfícies em lugar de apenas um vértice. O OpenGL oferece o sombreamento *flat* ou *smooth*. Um polígono numa superfície é designado de face. Seguem-se algumas considerações sobre a eficiência e qualidade do sombreamento das faces.

Back-Face Culling

Para acelerar o processo de desenho, podemos eliminar as superfícies escondidas antes do *rendering*. As faces posteriores não visíveis devem ser eliminadas o mais cedo possível do processo, antes mesmo do algoritmo *z-buffer* ser invocado. Em OpenGL, se a ordem dos vértices do polígono está no sentido horário quando este é visualizado a partir do ponto de vista, o polígono é designado de *front-facing*. Em caso oposto, ele é designado de *back-facing*. Podemos utilizar ***glEnable(GL_CULL_FACE)*** para activar a extracção dos polígonos *back-facing*. É importante garantir a ordem em que se especifica a lista de vértices, de forma a ser possível detectar quem está virado para frente ou para trás, de acordo com o ponto de vista. Efeitos indesejáveis podem ocorrer se isto não for bem feito. No exemplo ***J3_10_Lights.java*** é utilizado frequentemente o produto vectorial entre dois vectores-arestas para identificar a normal *n*.

```
void drawBottom(float *v1, float *v2, float *v3){
        // normal to the cone or cylinder bottom
        float v12[3], v23[3], vb[3];
        int i;
        for (i=0; i<3; i++) { // two edge vectors
                v12[i] = v2[i] - v1[i];
                v23[i] = v3[i] - v2[i];
        }
        // vb = normalized cross prod. of v12 X v23
        ncrossprod(v12, v23, vb);
        gl.glBegin(GL.GL_TRIANGLES);
                gl.glNormal3fv(vb);
                gl.glVertex3fv(v1);
                gl.glVertex3fv(v2);
                gl.glVertex3fv(v3);
        gl.glEnd();
}
```

Dada uma caixa oca ou cilindro sem tampa, deveremos ver tanto as faces com *back* e *front-culling*. Nesse caso, é possível activar a iluminação para ambas as faces: ***glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, TRUE)***. Se activarmos a iluminação nos dois lados, cada polígono tem dois lados com normais opostas, e nesse caso, o OpenGL irá decidir sombrear o lado que tem normal orientada para o ponto de vista. Podemos também facultar outras propriedades para o material de cada polígono *front-facing* ou *back-facing*: ***glMaterialfv(GL_FRONT, GL_AMBIENT, red)***; ***glMaterialfv(GL_BACK, GL_AMBIENT, green)***.

Modelos de sombreamento dos polígonos

A aparência de uma superfície debaixo de diferentes modelos de iluminação difere bastante. O sombreamento *flat*, que é o mais simples e rápido, é utilizado para mostrar

superfícies planas em lugar de curvas. Neste último caso, a aplicação de sombra *flat* numa malha poligonal mais pormenorizada, torna-se ineficiente e lento. O sombreamento *smooth* (também designada de *Gouraud*), calcula a cor com a interpolação da cor entre os *pixels* dos diversos vértices, e portanto, está mais orientado para superfícies curvas. Em OpenGL podemos utilizar ***glShadeModel(GL_FLAT)*** ou ***glShadeModel(GL_SMOOTH)*** para escolher entre os dois modelos de sombreamento. No OpenGL, os vértices das normais são especificados pelo programador. Para eliminar descontinuidades na intensidade, a normal a um vértice é calculada pela média das normais que partilhem o vértice na superfície. Em geral, se define a normal perpendicular a superfície curva e não ao polígono. Também é possível definir normais com outras direcções para se obter efeitos especiais.

Ray Tracing e Radiosidade

Estes dois aspectos são tópicos avançados na iluminação global e no processo de renderização. Ambos conferem maior realismo a cena gráfica. Como são relativamente complexos e exigentes a nível de processamento, não estão previstos pelo OpenGL.

Computação Gráfica (componente prática) - Parte 7

Temática: ***Desenho de curvas, superfícies curvas e sólidos***

Actividade 7: Utilização do JOGL para implementar superfícies quadráticas e modelos de objectos da GLUT, além de curvas e superfícies curvas

Competências a desenvolver:

- Conhecer as funcionalidades 3D incorporadas nas bibliotecas GLUT e GLU para o desenho de objectos sólidos.
- Saber utilizar o JOGL para criar programas que permitam desenhar curvas (Bézier e Hermite) e superfícies curvas (Bézier).

Curvas e Modelos Sólidos

Da mesma forma que existe vários métodos do tipo scan para desenhar primitivas gráficas, existem também inúmeras formas de criar modelos 3D. Por exemplo, podemos criar uma esfera através da subdivisão (como já referido em apontamento anterior). Podemos também utilizar a equação da esfera para encontrar os pontos e desenha-la. Além disso, podemos encontrar os pontos num círculo situado sobre o plano *xy* e gira-lo ao logo do eixo *x* ou *y*, obtendo assim, a esfera correspondente.

Embora a geração de modelos 3D não seja exactamente uma capacidade básica nos sistemas gráficos, é parte da teoria que suporta esta componente. Neste apontamento serão apresentados os modelos 3D e suas respectivas funções disponibilizadas nas bibliotecas GLUT e GLU. Ambas as bibliotecas facultam funções que podem desenhar o modelo de forma sólida ou em *wireframe* (armação em arame). O exemplo de código ***J5_1_Quadrics.java*** demonstra isso. Abaixo, o fragmento de código onde são utilizadas as funções apropriadas da GLUT e GLU para desenhar a esfera.

```
// Utilização da GLUT para desenhar a esfera
glutWireSphere(r, nLongitudes, nLatitudes);
glutSolidSphere(r, nLongitudes, nLatitudes);

// Utilização da GLU para desenhar a esfera
GLUquadric *sphere = gluNewQuadric();
gluQuadricDrawStyle(shpere, GLU_LINE); //GLU_FILL
glusphere(sphere, r, nLongitudes, nLatitudes);
```

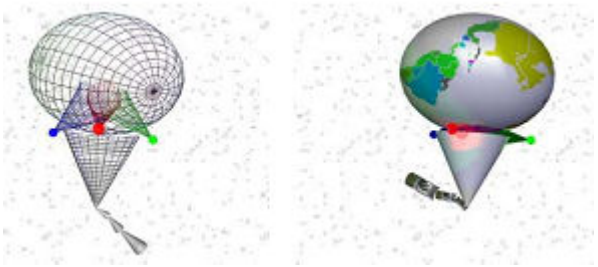
Da mesma forma, podemos encontrar todos os pontos de uma elipsóide com a sua equação. Outra maneira é através da alteração de escala de uma esfera – mais longa numa direcção que em outra. As bibliotecas GLU e GLUT não incluem o desenho automático de elipsóides, entretanto incluem o desenho cilindros também.

GLUT possui também funções para desenho de cones:

```
// Utilização de GLUT para desenhar um cone
glut.glutSolidCone(glu, r, h, nLongitudes, nLatitudes);
```

Mapeamento de textura nos modelos GLU

A GLU faculta a especificação automática de coordenadas de textura (imagens) para o render de seus modelos. A função utilizada é a ***gluQuadricTexture().*** Graças a isso, a aplicação de texturas nos modelos pré-definidos é muito simples. Só é necessário especificar os parâmetros da textura e dados como usual, não sendo necessário se preocupar como elas serão associadas as primitivas gráficas. No caso da GLUT, esse mecanismo só está disponível para o desenho do bule. A figura abaixo ilustra o desenho de sólidos em *wireframe* e com aplicação de textura.



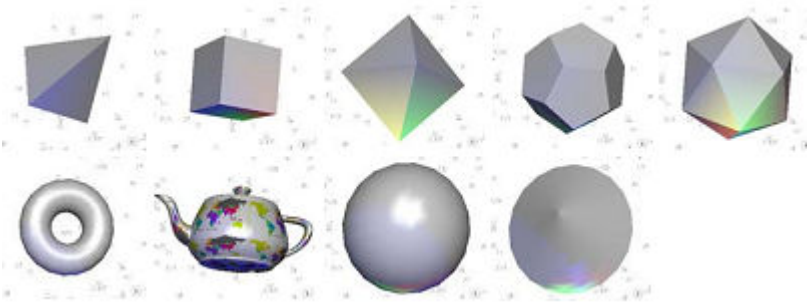
Torus, Poliedro e Bules na GLUT

Além do cone e da esfera, a GLUT faculta um conjunto de funções para desenhar modelos 3D, incluindo um toro, cubo, tetraedro, octaedro, dodecaedro, icosaedro e bule, tanto na forma sólida como em *wireframe*. Eles são fáceis de utilizar, como é demonstrado no programa ***J5_2_Solids.java***, onde é feita a substituição do desenho da esfera (***J5_1_Quadrics.java***) com diversos modelos 3D da GLUT.

Bules

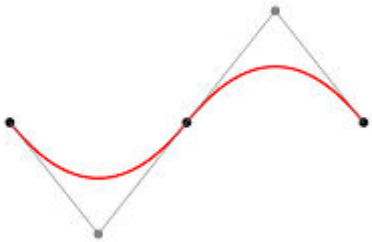
glutSolidTeapot() e ***glutWireTeapot()*** desenham um bule sólido ou em *wireframe*, respectivamente. Tanto as coordenadas para textura como as normais a superfície são geradas automaticamente pelo OpenGL. Tornando-se fácil o mapeamento de texturas como ilustrado na figura abaixo. De facto, este é o único modelo disponível na GLUT que está especificado de forma mais completa (normais e coordenadas de textura). O bule também pode ser obtido com outras funcionalidades do OpenGL – *evaluators* (será discutido mais a frente).

As primitivas da superfície do bule são todas *back-facing*. Isto é, todos os vértices do polígono estão especificados no sentido horário. Para executar o *back-face culling* é preciso indicar quem é a face frontal com ***glFrontFace(GL_CW)*** antes e desenhar o bule. Para voltar a situação normal deve ser utilizada a ***glFrontFace(GL_CCW)*** depois do bule ter sido desenhado. Como o bule é muito detalhado a nível de malha poligonal, é demorado o seu tempo de desenho (quando utilizam-se texturas).



Curvas e Superfícies Curvas

São vários os métodos matemáticos que existem para o desenho de curvas. O programa ***J5_3_Hermite.java*** ilustra o desenho de curvas utilizando o algoritmo de **Hermite**. No caso das curvas de **Bezier**, o OpenGL providencia um mecanismo básico designado de ***evaluators***. Ele utiliza a função ***glMap1f()*** para definir o intervalo (p. ex., 0≤ t ≤ 1), número de valores (p. ex., 3 para *xyz* ou 4 para *xyzw*) para ir de um ponto de controlo para o seguinte, grau da equação (p. ex., 4 para cúbica), e pontos de controlo (um *array* de pontos). Depois, em lugar de calcular os pontos da curva e utilizar ***glVertex()*** para especificar as coordenadas, é utilizado ***glEvaluCoord1()*** para especificar as coordenadas nos *t's*, e a curva de **Bezier** é calculada pelo sistema OpenGL como ilustrado no programa ***J5_4_Bezier.java.***



O OpenGL também fornece ***evaluators*** 2D para o desenho de superfícies **Bezier**. Esses ***evaluators*** também são um mecanismo básico para o desenho de superfícies de qualquer grau. Ele utiliza ***glMap2f()*** para definir o intervalo (p. ex., 0≤ *s*, *t* ≤1), número de valores em *s* ou direcções em *t* para chegar ao valor seguinte (p. ex., 3 para *xyz* ou 4 para *xyzw* na direcção *s*), grau da equação (p. ex., 4 para cúbica), e pontos de controlo (*array* de pontos). Então, em lugar de calcular os pontos da curva e utilizar ***glVertex()*** para especificar as coordenadas, é utilizado ***glEvaluCoord2(s, t)*** para especificar as coordenadas nas posições especificadas, e a superfície de **Bezier** é calculada pelo sistema OpenGL. No OpenGL, ***glMap*()*** também é utilizado para interpolar cores, normais e coordenadas de texturas.