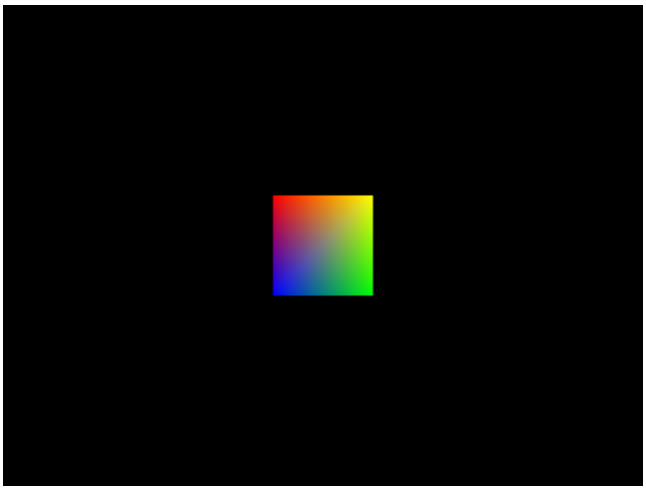
Lição 2: Matrizes e polígonos coloridos



Neste tutorial vamos criar e configurar um sistema de coordenadas de 640x480 e enquanto estivermos nele, vamos dar alguma cor ao nosso polígono. Também haverá uma breve explicação de como o OPenGL converte vértices em Pixels.

Os números de linhas usadas nas imagens servem apenas para identificação no corpo do texto. Elas podem variar em versões do código. Atente-se para isso.

Primeiro é necessário se importar todas as bibliotecas necessárias para o funcionamento do aplicativo.

```
 8  import java.util.logging.Level;
 9  import java.util.logging.Logger;
10  import org.lwjgl.LWJGLException;
11  import org.lwjgl.input.Keyboard;
12  import org.lwjgl.opengl.Display;
13  import org.lwjgl.opengl.DisplayMode;
14  import static org.lwjgl.opengl.GL11.GL_COLOR_BUFFER_BIT;
15  import static org.lwjgl.opengl.GL11.GL_MODELVIEW;
16  import static org.lwjgl.opengl.GL11.GL_NO_ERROR;
17  import static org.lwjgl.opengl.GL11.GL_PROJECTION;
18  import static org.lwjgl.opengl.GL11.GL_QUADS;
19  import static org.lwjgl.opengl.GL11.glBegin;
20  import static org.lwjgl.opengl.GL11.glClear;
21  import static org.lwjgl.opengl.GL11.glClearColor;
22  import static org.lwjgl.opengl.GL11.glColor3f;
23  import static org.lwjgl.opengl.GL11.glEnd;
24  import static org.lwjgl.opengl.GL11.glGetError;
25  import static org.lwjgl.opengl.GL11.glLoadIdentity;
26  import static org.lwjgl.opengl.GL11.glMatrixMode;
27  import static org.lwjgl.opengl.GL11.glOrtho;
28  import static org.lwjgl.opengl.GL11.glTranslatef;
29  import static org.lwjgl.opengl.GL11.glVertex2f;
```

Faz se necessário entender que o uso do caractere coringa (*) para importar tudo, não funcionou corretamente. Dessa forma, aconselhamos que se faça todas as importações individualmente.

Acima, podemos ver praticamente todas as funções que vamos utilizar: as funções das linhas 8 e 9 tratam dos métodos no qual o Java trata métodos de funcionamento da linguagem.

Das linhas 10 até 13, temos métodos de entrada e saída, com métodos que usamos para o teclado (no qual vamos pegar teclas) e o monitor (no qual vamos mostrar o polígono) e as exceções específicas da Biblioteca LWJGL.

Das linhas 14 até 29, temos todas os métodos de OpenGL propriamente ditos. Esses métodos podem ser usados em qualquer linguagem sem alteração, entretanto, os métodos das linhas 8 até 16 são exclusivos de JAVA (e da Biblioteca LWJGL). A explicação detalhada de cada método será dada quando o método for utilizado.

1. A Classe Main

Para inicializar o aplicativo, vamos criar uma classe MAIN, que conterá o método *main* com a chamada dos métodos que efetivamente *fazem o trabalho* e algumas variáveis globais necessárias no aplicativo.

```
35  public class MainLição02 {
36      public static final int DISPLAY_HEIGHT = 480;
37      public static final int DISPLAY_WIDTH = 640;
38      public static final Logger LOGGER = Logger.getLogger(MainLição02.class.getName());
39      public int gColorMode = 0; //0 para CYAN e 1 para MULTI
40      public float gProjectionScale = 1.f;
41
42      public static void main(String[] args) {
            MainLição02 main = null;
44          try {
45            main = new MainLição02();
46            main.create();
47            main.run();
48          }
            catch(Exception ex) {
50            LOGGER.log(Level.SEVERE,ex.toString(),ex);
51          }
52          /*finally {
53            if(main != null) {
54              main.destroy();
55            }
56          }*/
57      }
```

1.1. Linhas 35 a 40

As variáveis globais aqui definem o comportamento de algumas coisas no aplicativo. Precisamos, primeiro, definir o tamanho da janela que queremos criar. Aqui, guardamos esses valores em *DISPLAY_HEIGHT* e *DISPLAY_WIDTH*, sendo o primeiro a altura e o segundo a largura. Como esses valores não são alterados, podemos defini-los como finais.

Logo após temos a criação do LOGGER, uma variável que vai conter o logger do que acontecer no programa.

A linha 42 define que a primeira cor no qual o nosso polígono vai ter é a CIANO (*CYAN),* e para isso usamos o valor inteiro 0 para defini-la. Caso quisermos usar multi cores, usamos 1. A linha seguinte define a escala. A escala, geralmente, é definida na maneira *valor_da_escala.f*, sendo 1 o tamanho normal, e variações, abaixo ou acima, multiplicações dessas escalas.

1.2. Linhas 42 a 57

A chamada do método *main* do Java contém as criação da janela que vamos desenhar nosso polígono e a execução do OpenGL dentro dessa janela, bem como o tratamento da exceção que eventuais erros possam criar.

Primeiro definimos uma instancia da classe principal (aqui nomeada de “MainLição02”) e definimos ela como *null* (linha 43) . Logo após, o try-catch tenta executar o aplicativo. Perceba que as únicas chamadas nesse *try* são *main.create( )*, que cria uma janela (que será explicado mais tarde) e *main.run( )*, que efetivamente desenha o polígono.

2. O método *create( )*

```
85      public void create() throws LWJGLException {
86          //Display
87          Display.setDisplayMode(new DisplayMode(DISPLAY_WIDTH,DISPLAY_HEIGHT));
88          Display.setFullscreen(false);
89          Display.setTitle("Lição 02");
90          Display.create();
91
92          //Keyboard
93          Keyboard.create();
94
95          //OpenGL
96          initGL();
97      }
```

Esse método é o responsável por inicializar o ambiente, criando uma janela para que o polígono seja exibido e criar a entrada do teclado, bem como iniciar o OpenGL.

2.1. Criando e configurando a janela

Nossa janela é criada nas linhas 87 a 90. A Classe *Display* utilizada faz parte da biblioteca LWJGL. Ela define:

- O tamanho da janela, na linha 87. Nela podemos passar valores ou apenas campos da classe principal, como no código. Essa linha é responsável por criar a instancia de saída que iremos usar, com o uso do método *setDisplayMode*, que recebe uma nova instancia de *DisplayMode* com dois valores: a largura (width) e a altura (height) da nossa janela.

- A linha 88 define se queremos que, ao iniciar, o aplicativo funcione em tela cheia. Nesse caso, não queremos. Assim, vamos definir como *false*.
- A linha 89 define um título para nossa janela.
- E por fim, o método *Display.create( ),* que pega essas informações e cria a janela, para que o OpenGL.

2.2. O teclado

A linha 93 cria uma instancia para a criação da entrada padrão do teclado. O método não tem entrada nem retorno.

2.3. initGL( )

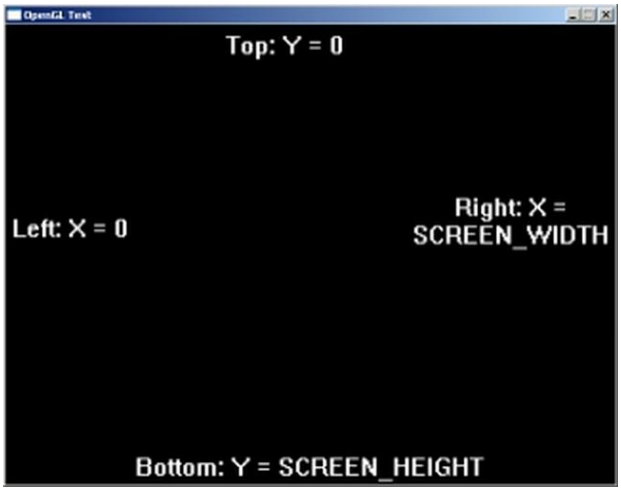
Este método está definido dentro da classe. Ele é o responsável por iniciar todos os campos e diretrizes. Ele tem um retorno boleano: caso tudo possa ser inicializado, ele restorna *true*, caso contrário, ele mostra o erro e retorna *false.*

```
79      public boolean initGL() {
80          //Iniciar projeção da matriz
81          glMatrixMode(GL_PROJECTION);
82          glLoadIdentity();
83          glOrtho(0.0, DISPLAY_WIDTH, DISPLAY_HEIGHT, 0.0, 1.0, -1.0);
84
85          //Inicializar ModelView da Matrix
86          glMatrixMode(GL_MODELVIEW);
87          glLoadIdentity();
88
89          //Iniciar cor
90          glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
91
92          //Checagem de erros
93          int erro = glGetError();
94          if(erro != GL_NO_ERROR)
95          {
96              System.out.println("Erro de inicialização do OpenGL");
97              return false;
98          }
99          return true;
100     }
```

2.3.1. Linhas 80 a 83

Na linha 81 definimos a matriz no modo *Projeção* e depois colocamos nela uma matriz identidade (linha 82).

Na linha 83 definimos essa matriz como ortogonal e passamos para ela como queremos nossa matriz. O mais importante a se notar aqui, é que por ser uma matriz tridimensional, temos que colocar os valores que façam com que nossa matriz tenha o tamanho da nossa janela.



A imagem mostra os 4 primeiros valores do método *glOrtho*, sendo os dois últimos valores de *near* e *far.*

2.3.2. Linhas 86 e 87

Agora temos que inserir a mesma matriz identidade na matriz *MODELVIEW*. Para isso, alteramos a matriz que estamos mexendo na linha 86 de *GL_PROJECTION* para *GL_MODELVIEW* e na linha seguinte, carregamos a matriz identidade.

2.3.3. Limpando cores

O método glClearColor recebe 4 parâmetros (de ponto flutuante) e limpa as cores: o primeiro vermelho, o segundo verde, o terceiro azul, e o quarto o Alpha (a opacidade das cores).

3. O método *RUN( )*

Após criada a janela, devemos agora fazer com que ela receba os comandos OpenGL que precisa para desenhar o que desejamos. Nesse caso, desenharemos um quadrado, que ao pressionar *E* ele se escalona em 3 níveis e ao apertar *Q* ele muda de cor.

```
170     public void run() throws LWJGLException, InterruptedException {
171
172         while(!Display.isCloseRequested()) {
173             //System.out.println(Display.isVisible());
174             if(Display.isVisible()){
175               processKeyboard();
176               render();
177               update();
178             }
179             else {
180               if(Display.isDirty()) {
181                 render();
182               }
183               try {
185                 Thread.sleep(100);
186               }
187               catch(InterruptedException ex) {
188               }
189             }
190             Display.sync(10);
191             Display.update();
192         }
    }
```

Este método pode lançar uma exceção, caso haja erros, para o método que o chamou (*main)* e não tem retorno. Todo o conteúdo do else é tratamento do JAVA, em caso de erro.

As linhas 189 e 190 definem o tempo sincronização em milisegundos e a atualização (*update*) da tela. O Método *update*, nesse aplicativo, está vazio, pois não temos muitas interações nem movimento.

3.1. O LOOP WHILE

Este loop está presente em praticamente todas as aplicações de OpenGL. Dentro dele, colocamos as rotinas de OpenGL que queremos que o monitor mostre e as mantemos repetindo até que a janela feche.

Quando a janela é fechada, há o pedido para isso. Nesse caso, podemos ver dentro do loop *while* que enquanto não for solicitado ao *Display* que o mesmo se encerre, o loop deve continuar.

3.1.1. ProcessKeyboard

Aqui, temos um método que vai coletar do teclado os comandos para alterar o comportamento do que é desenhado na tela.

```
102     public void processKeyboard() {
103         //Se pressionar a letra Q
104         if(Keyboard.isKeyDown(Keyboard.KEY_Q))
105         {
106             //Mudando a cor
107             if(gColorMode == 0) //CYAN
108                 gColorMode = 1; //MULTI
109             else
110                 gColorMode = 0; //CYAN
111         }
112         //Se pressionar a letra E
113         else if (Keyboard.isKeyDown(Keyboard.KEY_E))
114         {
115             //Muda as escalas
116             if(gProjectionScale == 1.f)
117                 gProjectionScale = 2.f; //Mais zoom
118             else if(gProjectionScale == 2.f)
119                 gProjectionScale = 0.5f; //Menos zoom
120             else if(gProjectionScale == 0.5f)
121                 gProjectionScale = 1.f; //Tamanho normal
122         }
123 
124         //Atualizando a projeção da matrix
125         glMatrixMode(GL_PROJECTION);
126         glLoadIdentity();
127         glOrtho(0.0, DISPLAY_WIDTH * gProjectionScale, DISPLAY_HEIGHT * gProjectionScale, 0.0, 1.0, -1.0);
128     }
```

Podemos ver no código duas coisas:

- Nas linhas 104 até 111, vemos que se o usuário digitar a letra *Q* as cores mudam. Podemos ver que o *if* confere a cor atual e a muda, com o pressionar da tecla. Aqui, temos apenas dois modos: ciano e multicolorido. Essas linhas alteram diretamente o campo gColorMode, presente no *Main* da classe.
- Nas linhas 113 até 122, podemos ver que, na mesma forma, o campo gProjectionScale é alterado quando pressionado o E. O que ele faz é alterar o tamanho escalar dos valores na matriz do display, multiplicando pelo valor dado.

Após fazer a alteração, as linhas 125 a 127 fazem o calculo para alterar a projeção da matriz de projeção, multiplicando a largura e a altura pela escala:

- Escala de 100%: padrão

- Escala de 200%: nós renderizamos uma área duas vezes maior, então tudo é menor.
- Escala de 50%: nós renderizamos uma área metade do total, assim, tudo parece maior.

Depois que a escala é selecionada, nós redefinimos a matriz identidade da matriz de projeção e multiplicamos com a matriz de perspectiva ortográfica em escala.

Aqueles de vocês que realmente conhecem a sua álgebra linear provavelmente estão pensando: "Não podemos simplesmente multiplicar uma matriz de escala contra a matriz de projeção?". A resposta é sim, e estaremos cobrindo a transformação para a matriz de projeção no tutorial de rolagem.

3.1.2. *render( )*

O método *render* é o responsável por desenhar, efetivamente, o polígono na tela.

```
130     public void render() throws LWJGLException {
131         //Clear color buffer
132         glClear(GL_COLOR_BUFFER_BIT);
133 
134         //Reset ModelView Matrix
135         glMatrixMode(GL_MODELVIEW);
136         glLoadIdentity();
137 
138         //Move to center of the display
139         glTranslatef(DISPLAY_WIDTH / 2.f, DISPLAY_HEIGHT / 2.f, 0.f);
140 
141         //Render quad
142         if(gColorMode == 0) //CYAN
143         {
144             //Solid Cyan
145             glBegin(GL_QUADS);
146                 glColor3f(0.f, 1.f, 1.f);
147                 glVertex2f(-50.f, -50.f);
148                 glVertex2f(50.f, -50.f);
149                 glVertex2f(50.f, 50.f);
150                 glVertex2f(-50.f, 50.f);
151             glEnd();
152         }
153         else
154         {
155             //RYGB Mix
156             glBegin(GL_QUADS);
157                 glColor3f(1.f, 0.f, 0.f);
158                 glVertex2f(-50.f, -50.f);
159                 glColor3f(1.f, 1.f, 0.f);
160                 glVertex2f(50.f, -50.f);
161                 glColor3f(0.f, 1.f, 0.f);
162                 glVertex2f(50.f, 50.f);
163                 glColor3f(0.f, 0.f, 1.f);
164                 glVertex2f(-50.f, 50.f);
165             glEnd();
166         }
167         Display.swapBuffers();
168     }
169 
```

Aqui o nosso método de renderização. Depois de limpar a tela, definimos o modo de matriz atual para *modelview*. Fazemos isso porque em nosso método de manipulação de teclas vamos mudar a matriz de projeção. Se não tivermos certeza de que a matriz atual é a matriz *modelview*, as operações de matriz de projeção e *modelview* serão feitas incorretamente e obteremos resultados indesejados.

A razão pela qual precisamos da matriz *modelview* é porque vamos aplicar transformações à nossa geometria. Quando a matriz de projeção controla como a geometria é visualizada, as transformações de matriz modelview controlam como a geometria é colocada no mundo de renderização.

Devido à forma como configuramos nossa matriz de projeção, a origem da cena está no canto superior esquerdo. Queremos que o nosso quadrado apareça no meio da tela, então nós a transladamos (ou a deslizamos) para o centro da tela usando *glTranslate ( )*. *glTranslate ( )* multiplica uma matriz de

translação contra a matriz atual (que no caso é a matriz *modelview*) de modo que qualquer geometria que é processada é traduzida a quantidade x, y e z dada nos argumentos.

Outra coisa a ser observada é que em cada frame carregamos a matriz identidade do *modelview* antes da renderização. Se não o fizéssemos, as transformações de translação se acumulariam. Então, se transladássemos 10 quadros, depois de 60 quadros, teríamos transladado 600. Eventualmente, nós transladaríamos completamente da tela.

```
132             glClear(GL_COLOR_BUFFER_BIT);
133
134             //Reset ModelView Matrix
135             glMatrixMode(GL_MODELVIEW);
136             glLoadIdentity();
137
138             //Move to center of the display
139             glTranslatef(DISPLAY_WIDTH / 2.f, DISPLAY_HEIGHT / 2.f, 0.f);
```

A partir da linha 141, desenhamos nosso polígono. Se o modo de cor é ciano (que é o valor inicial), queremos renderizar um quad de cor sólida.

Para definir a próxima cor de vértice, chamamos glColor () com os valores de vermelho, verde e azul. Em seguida, definimos a posição do vértice com glVertex ().

Pela segunda vez que enviamos um vértice (bem como o terceiro e quarto), não damos cor. Portanto, o pipeline OpenGL apenas usa o último valor de cor que foi dado. Isso dá ao nosso quad uma cor ciana sólida.

Você notará que estamos fazendo um quadrado com uma largura de 100. Graças à nossa matriz de projeção, ela deve ter 100 pixels de largura.

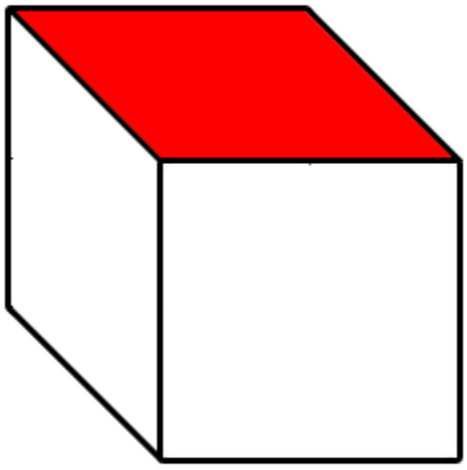
```
142             if(gColorMode == 0) //CYAN
143             {
144                 //Solid Cyan
145                 glBegin(GL_QUADS);
146                     glColor3f(0.f, 1.f, 1.f);
147                     glVertex2f(-50.f, -50.f);
148                     glVertex2f(50.f, -50.f);
149                     glVertex2f(50.f, 50.f);
150                     glVertex2f(-50.f, 50.f);
151                 glEnd();
152             }
```

Se "gColorMode" não é ciano, nós assumimos que ele deve ser multicor.

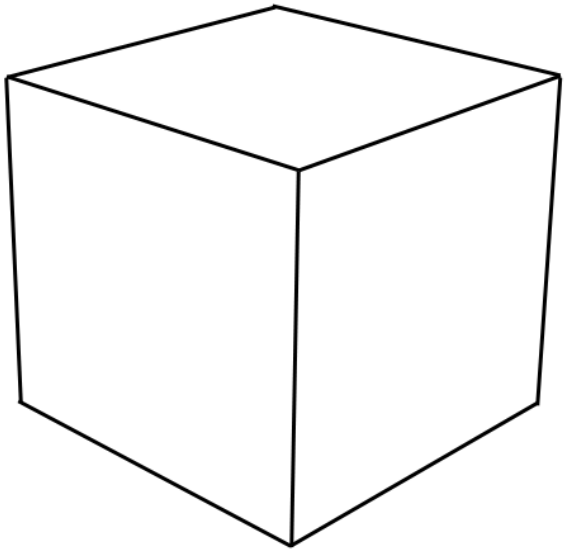
```
155             //RYGB Mix
156             glBegin(GL_QUADS);
157                 glColor3f(1.f, 0.f, 0.f);
158                 glVertex2f(-50.f, -50.f);
159                 glColor3f(1.f, 1.f, 0.f);
160                 glVertex2f(50.f, -50.f);
161                 glColor3f(0.f, 1.f, 0.f);
162                 glVertex2f(50.f, 50.f);
163                 glColor3f(0.f, 0.f, 1.f);
164                 glVertex2f(-50.f, 50.f);
165             glEnd();
166         }
```

Observe que desta vez, damos uma cor individual para cada vértice. Novamente, é importante dar a cor antes de dar o vértice porque OpenGL olha para o valor de cor mais recente ao renderizar esse vértice particular.

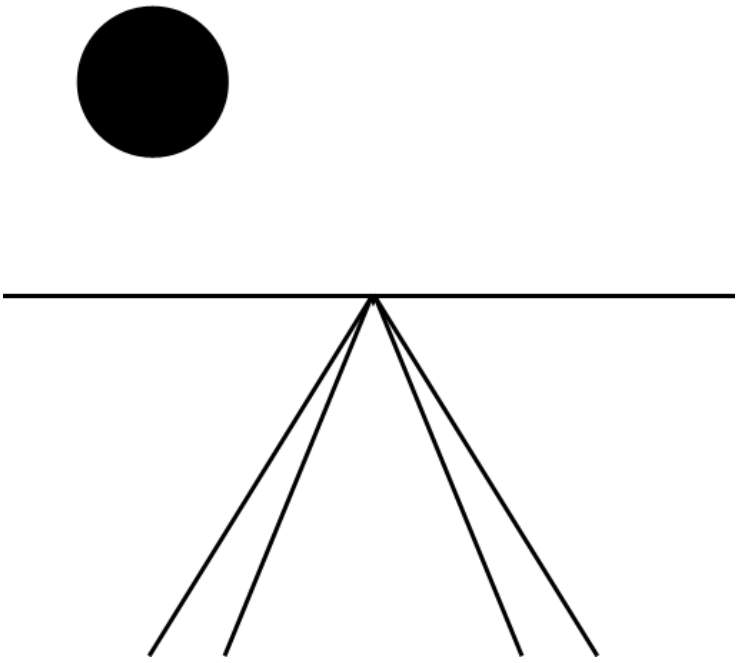
Ok, então nós rendemos nosso quad. Agora, como o OpenGL usa matrizes para transformar polígonos em pixels? Vamos usar um polígono de um cubo 3D como um exemplo. Aqui está um cubo (mal desenhado):



Em primeiro lugar, a matriz de *modelview* é aplicada para transladá-lo / rodá-lo / dimensioná-lo / incliná-lo / tudo para o lugar:

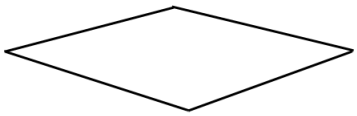


Lembre-se de volta à aula de arte do ensino médio e aquelas cenas de perspectiva que eles fizeram você fazer?

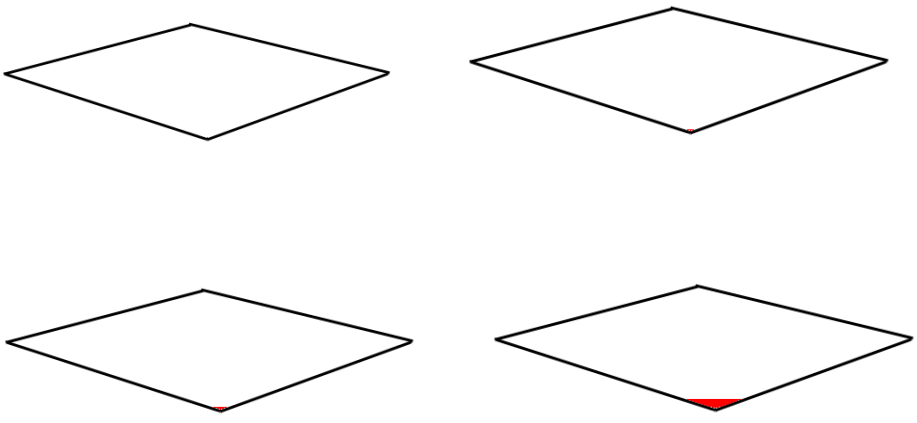


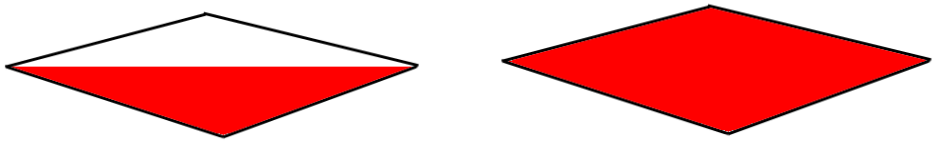
O que a matriz de projeção faz é tomar seus vértices de seu polígono e os multiplica para transformá-los em coordenadas de perspectiva normalizadas que o OpenGL pode usar:

Em seguida, ele conecta seus vértices do polígono...



E começa a preencher os pixels (isso é chamado de rasterização):





Obviamente, há mais no pipeline do OpenGL com coisas para controlar a texturização, coloração, iluminação, etc. Em termos de como chegamos da geometria aos pixels, tudo que o OpenGL faz é tomar coordenadas de vértices e rasterizá-los em pixels com o ProjectionMatrix * ModelviewMatrix * Vertex.

```
167             Display.swapBuffers();
```

No final do nosso método *render( )* depois que nossos vértices foram rasterizados em pixels, trocamos o buffer frontal / traseiro para atualizar a tela.

A partir desse momento, métodos utilizados e já explicados nessa lição não serão explicados novamente. Apenas códigos novos e inéditos serão abordados nas lições.