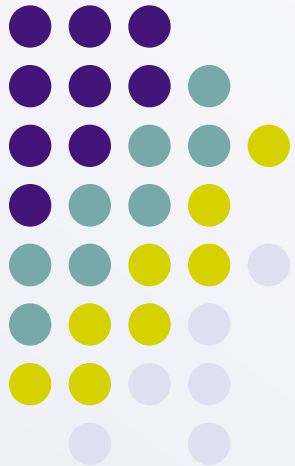
Introdução à Computação Gráfica

Marcel P. Jackowski
mjack@ime.usp.br

Aula #3: Noções básicas de visualização

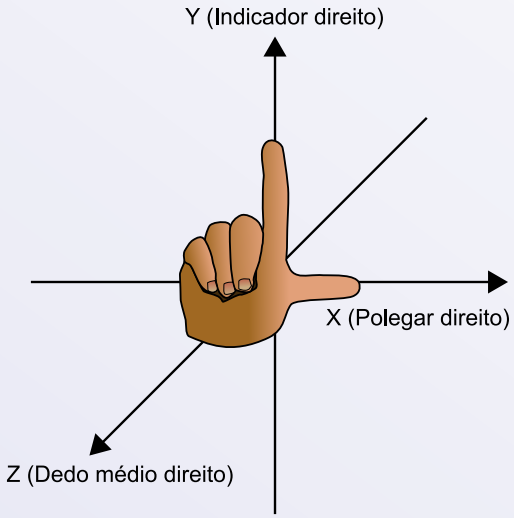


Objetivos

- Visualização em 2D
  - Triângulo de Sierpinski
  - Modificando viewports
  - Interação via mouse
- Visualização em 3D
  - Sierpinski em 3D
  - Transformações
- Tarefa #3

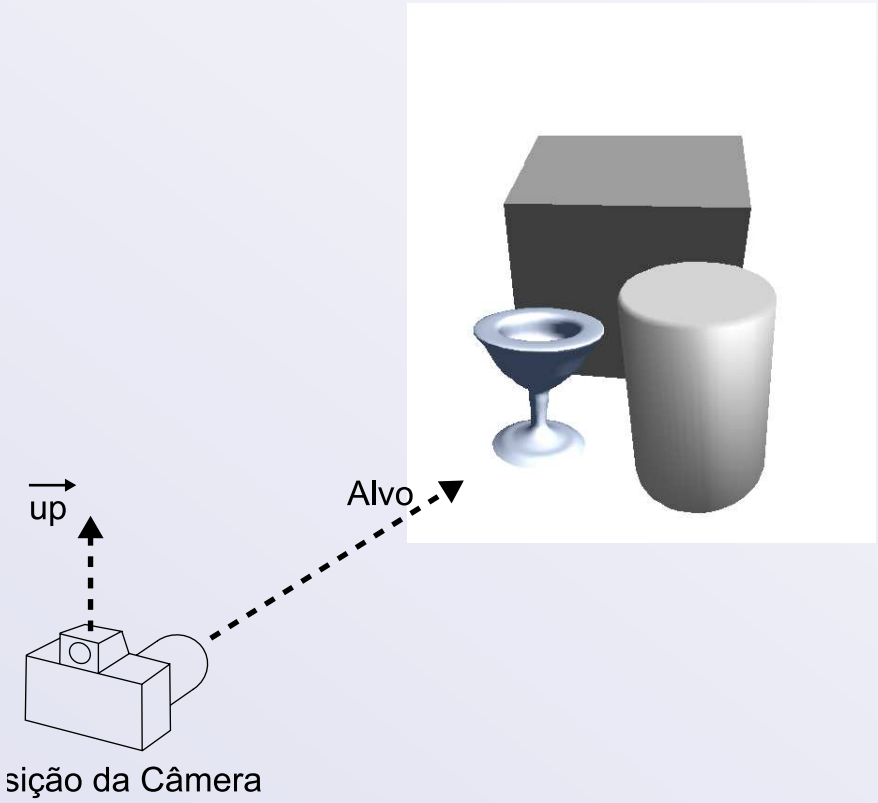
Sistemas de coordenadas

- As unidades de **`glVertex`** são determinadas pela aplicação, e são chamadas de *coordenadas do sistema de referência do universo* (SRU)

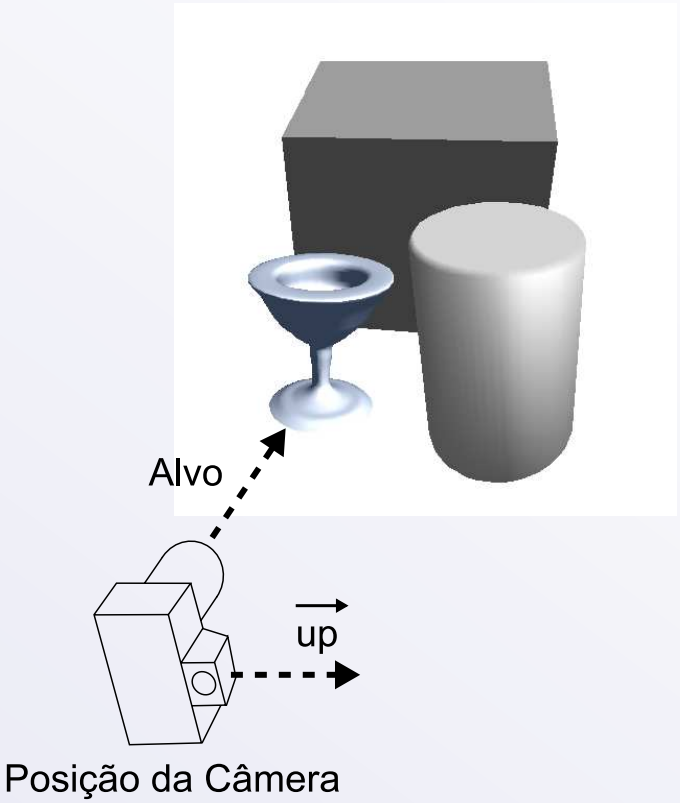


- As especificações de visualização também são em coordenadas do universo. O tamanho do *volume de visualização* ("view volume") determina o que aparece na imagem
- Internamente, o OpenGL converterá tais valores em *coordenadas da câmera* e depois as converterá para *coordenadas da tela (janela)*

Câmera



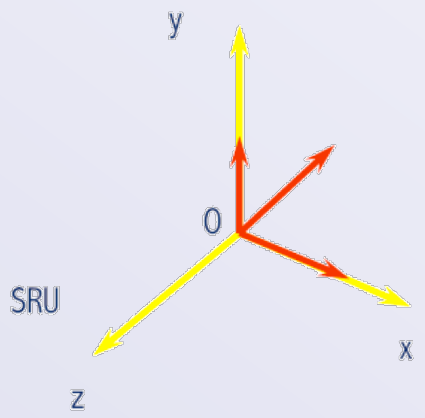
(a) Câmera na orientação normal

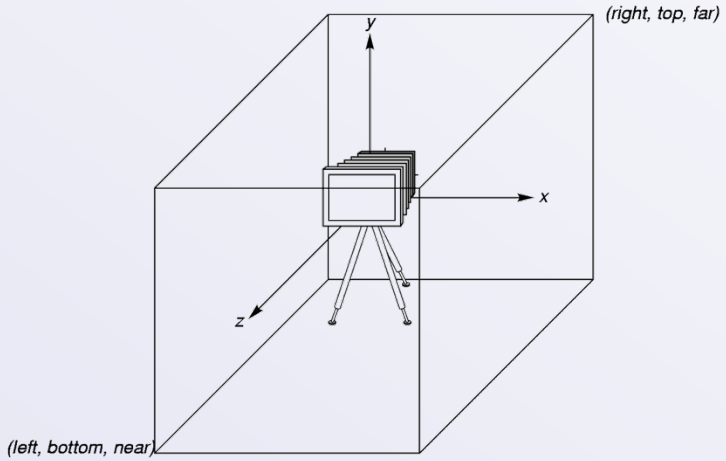


(b) Câmera inclinada

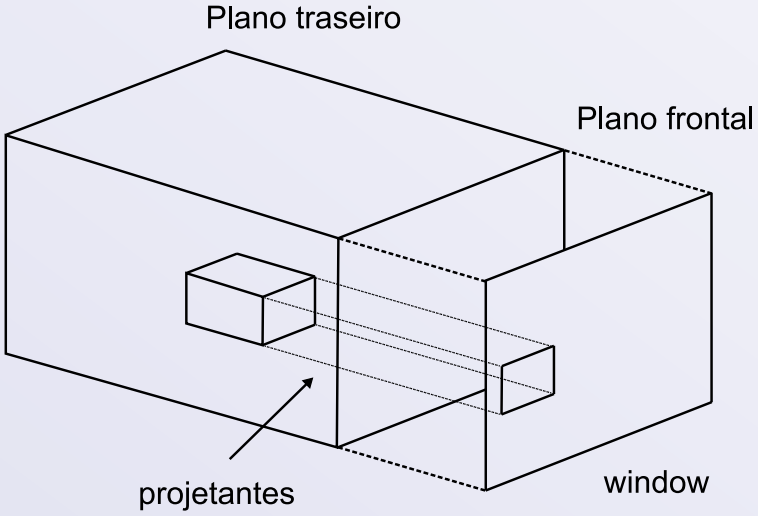
A câmera em OpenGL

- OpenGL posiciona a câmera na origem do espaço dos objetos apontando na direção negativa de z
- O volume de visualização default é um cubo centrado na origem com lado medindo 2.

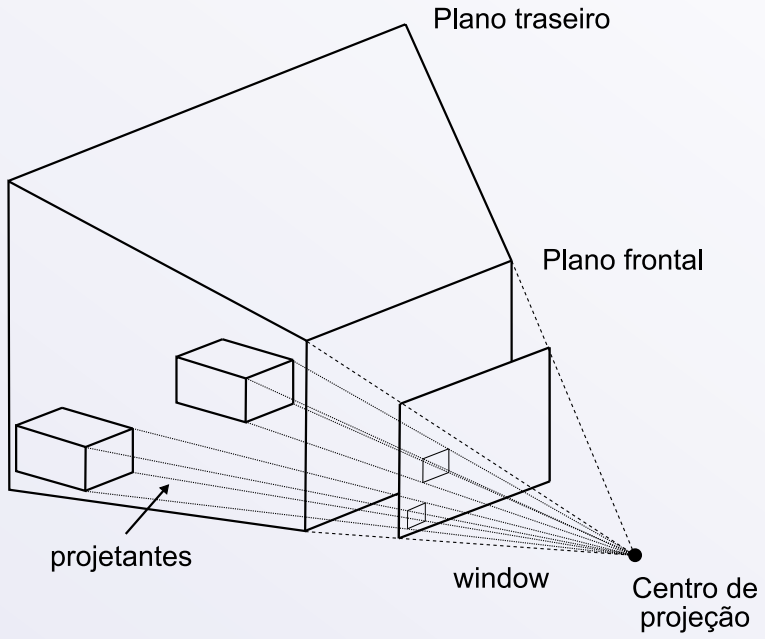




Projeções



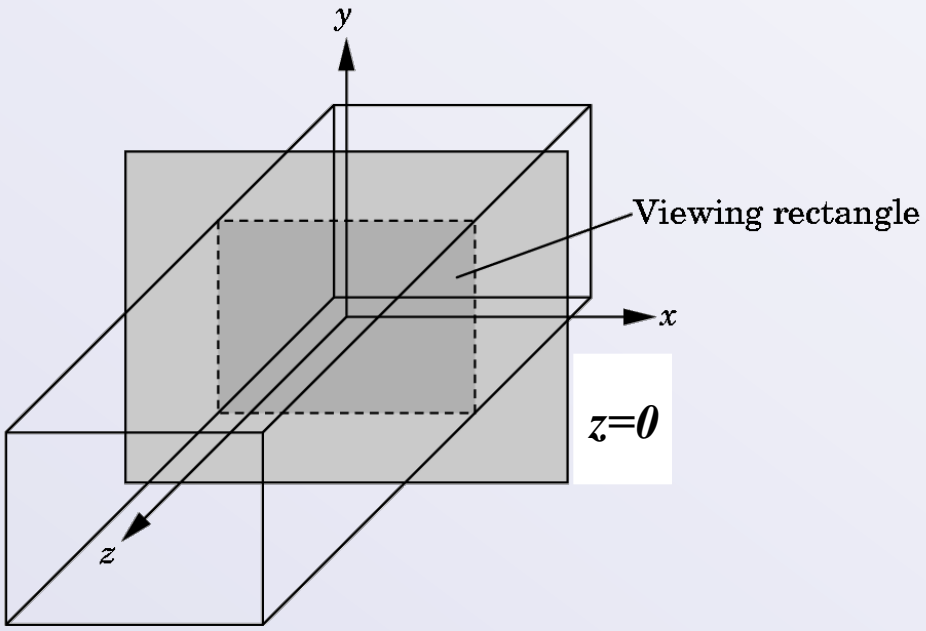
(a)

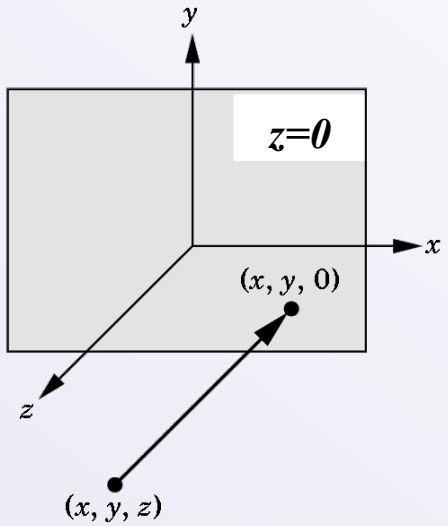


(b)

Projeção ortográfica em OpenGL

- Na projeção ortográfica, pontos são projetados ao longo do eixo *z* no plano definido por *z=0*





Transformações de visualização

- Projeções são implementadas por matrizes (transformações)
- Existe somente um único conjunto de funções de transformação

```
glMatrixMode(GL_PROJECTION);
```

- Tais funções operam de modo incremental
  - Primeiramente, começamos com a matriz identidade
  - Alteramos com a matriz de projeção que define o volume de visualização

```
glLoadIdentity();
glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```

Visualização 2D e 3D

- Em **`glOrtho(left, right, bottom, top, near, far)`** e as distâncias “near” e “far” são medidas a partir da câmera
- Vértices em 2D são posicionados no plano z=0
- Caso uma aplicação opere em 2D, podemos usar a função

  `gluOrtho2D(left,right,bottom,top)`
- Em 2D, o volume de visualização se torna uma “janela” de visualização

simple1.c

```
#include <GL/glut.h>

int main(int argc, char** argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(500,500);
   glutInitWindowPosition(0,0);
   glutCreateWindow("simple");
   glutDisplayFunc(mydisplay);

   init();

   glutMainLoop();
}
```

inclui **gl.h**

define propriedades da janela

callback de display

inicializa estado do OpenGL

entra laço de eventos

Função `init()`

```
void init()
{
   glClearColor (0.0, 0.0, 0.0, 1.0);

   glColor3f(1.0, 1.0, 1.0);

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
}
```

cor preta

opaca

seleciona cor branca

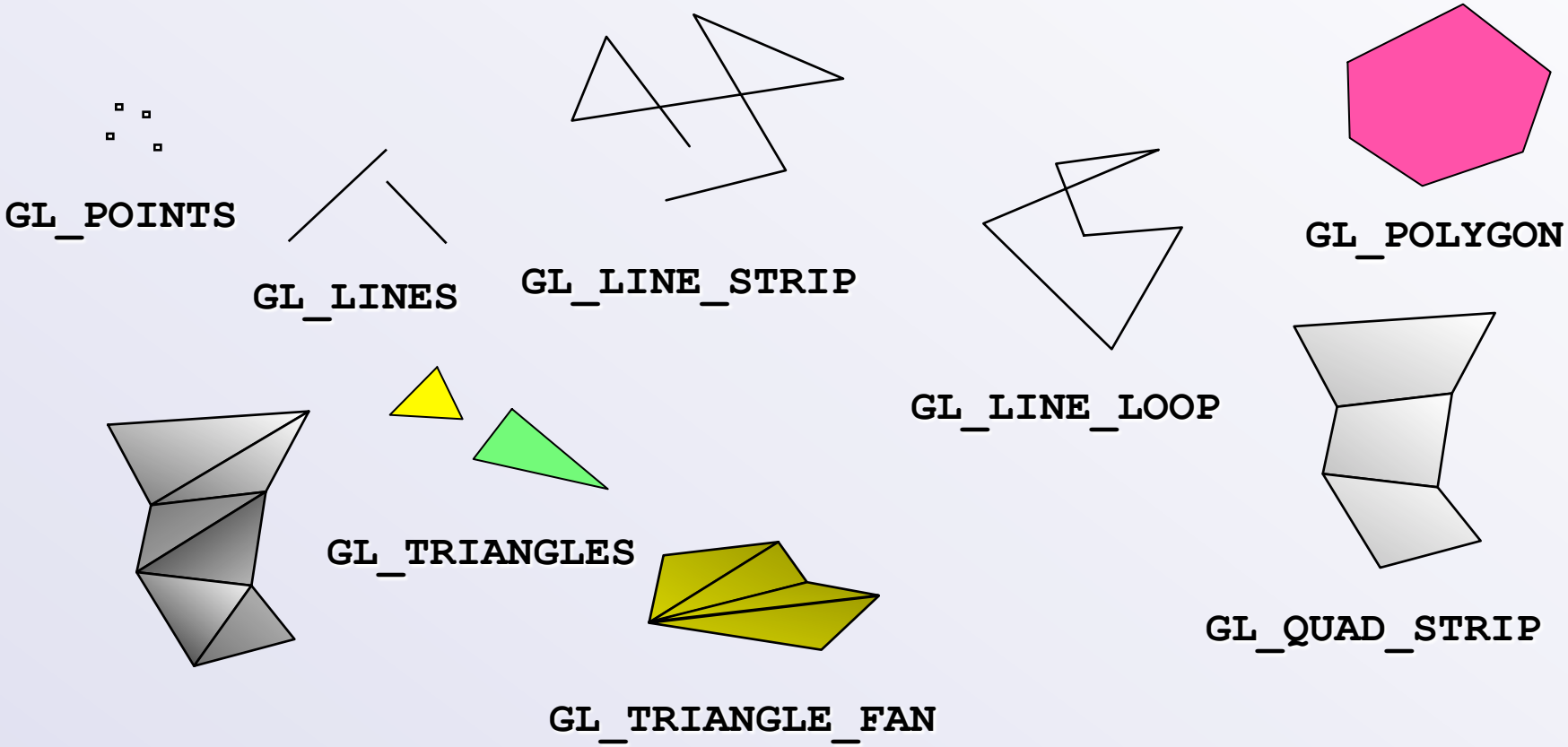
define volume de visualização

mydisplay

```
void mydisplay()
{
  glClear(GL_COLOR_BUFFER_BIT);
  glBegin(GL_POLYGON);
     glVertex2f(-0.5, -0.5);
     glVertex2f(-0.5, 0.5);
     glVertex2f(0.5, 0.5);
     glVertex2f(0.5, -0.5);
  glEnd();
  glFlush();
}
```

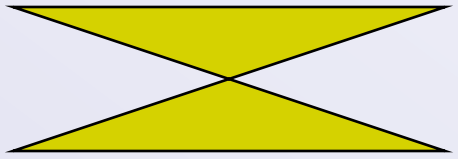
Demo simple1.c

Primitivas geométricas



Triângulos

- OpenGL somente desenhará corretamente polígonos que são:
  - Simples: arestas não se cruzam
  - Convexos: todos os pontos em um segmento de linha entre dois pontos dentro do polígono também estão dentro do polígono
  - Planares: todos os vértices estão no mesmo plano
- Programa deverá verificar estas condições
  - OpenGL produzirá uma saída mesmo se tais condições forem violadas
- Triângulos satisfazem todas as condições



Polígono complexo

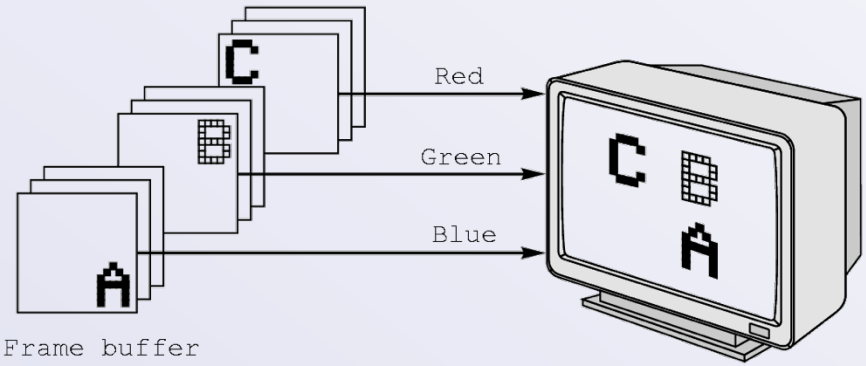
Polígono côncavo

Atributos

- Attributos são parte do estado OpenGL e determinam a aparência dos objetos
  - Cor (pontos, linhas, polígonos)
  - Tamanho (pontos, linhas)
  - Padrão de “stipple” (linhas, polígonos)
  - Modo de exibição de polígonos
    - Modo Preenchido (cor sólida ou padrão stipple)
    - Modo Arestas (wireframe)
    - Modo Vértices (pontos)

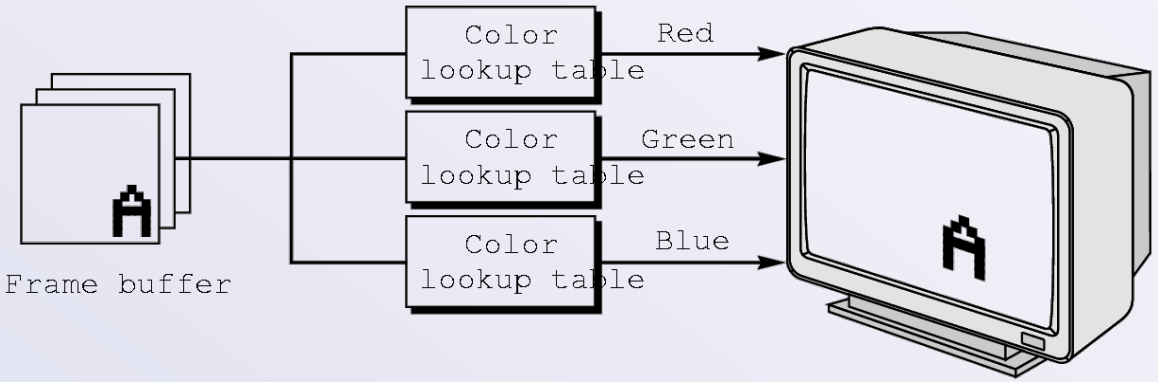
Cores

- Cada componente de cor é armazenado individualmente no frame buffer
- Normalmente existem 8 bits por componente no buffer
- Note que em **`glColor3f`** os valores variam de 0.0 a 1.0, e em **`glColor3ub`** variam de 0 a 255



Cores indexadas

- Cores se tornam índices para tabelas contendo valores RGB
- Requer menos memória
  - Índices são normalmente 8 bits
  - Hoje em dia, não tão importantes
    - Memória mais barata
    - Necessidade de uma paleta maior para tonalização



Cores e estados

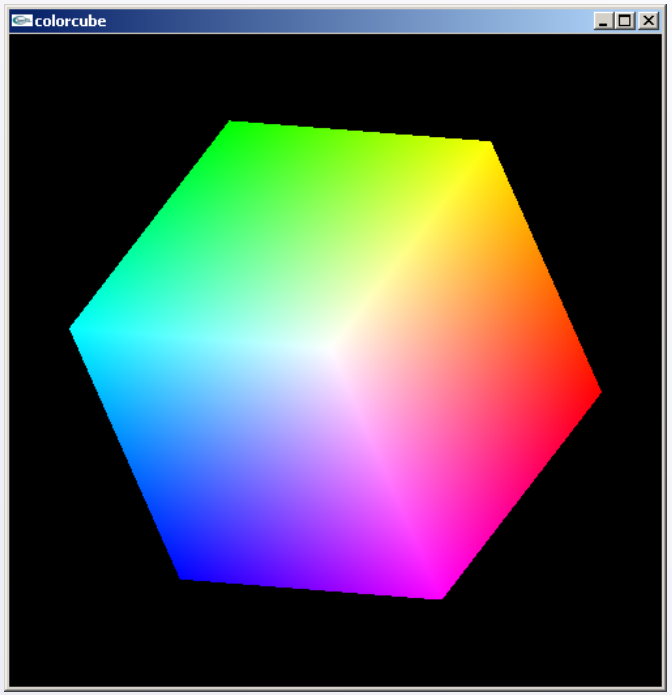
- A cor selecionada por **`glColor`** permanece parte do estado e continuará a ser usada até que seja alterada novamente
  - Cores e outros atributos não são partes do objeto em si, mas são atribuídos quando o objeto é renderizado
- Podemos atribuir diferentes cores para diferentes vértices da seguinte forma:

```
glColor
glVertex
glColor
glVertex
```

Tonalização

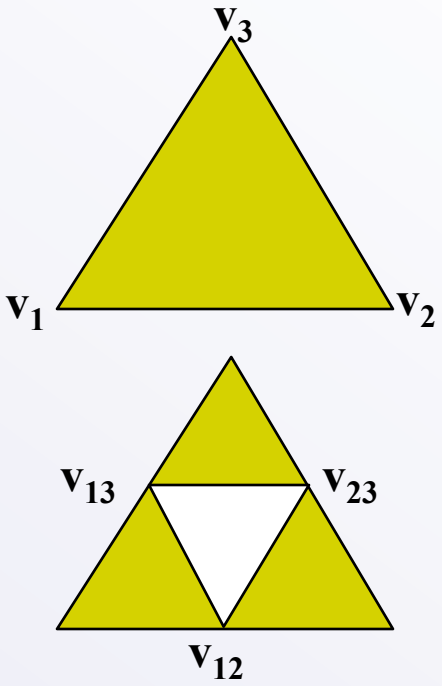
- O default é a *tonalização suave*
  - OpenGL interpola valores dos vértices ao longo dos polígonos visíveis
- Alternativa é a *tonalização facetada*
  - Cor do primeiro vértice determina a cor de preenchimento
- **`glShadeModel`**

  `(GL_SMOOTH` ou **`GL_FLAT)`**



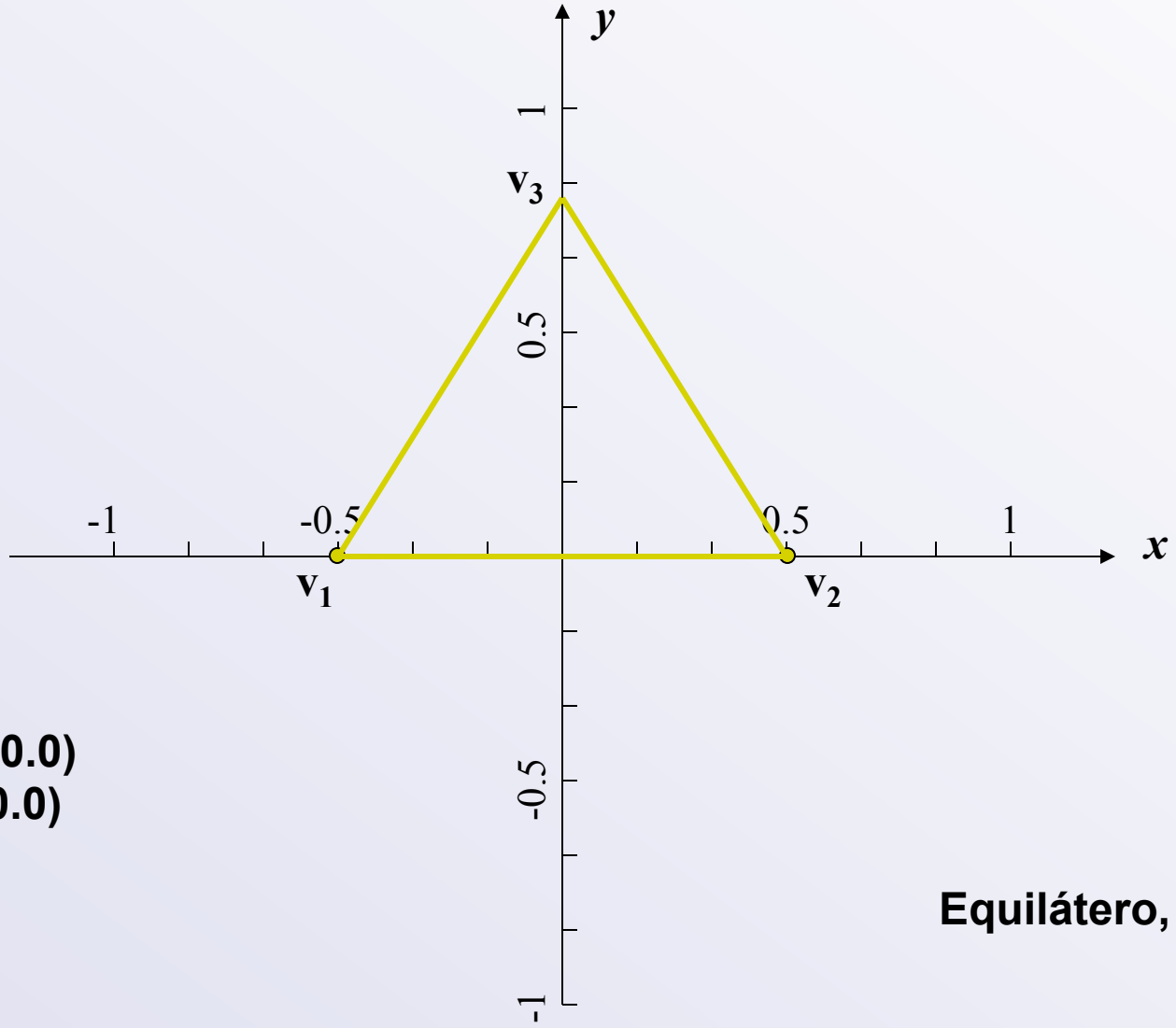
Fractal: Triângulo de Sierpinski

- Inicia-se com um triângulo
- Conecta-se bisetores dos lados e remove-se o triângulo central
- Repete-se o processo



Waclaw Sierpinski (1882 - 1969), matemático polonês

Coordenadas do triângulo



v_1= (-0.5,0.0)
v_2= (0.5,0.0)
v_3=?

Equilátero, lado = 1cm

Sierpinski2D.c

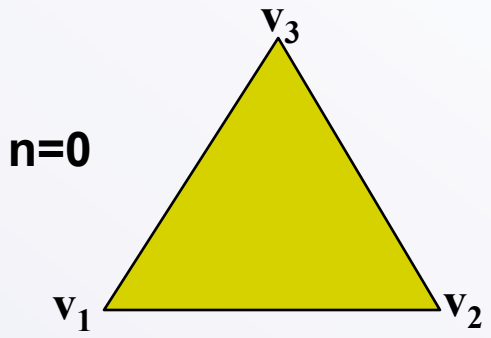
```
#include <GL/glut.h>

/* triângulo inicial */
GLfloat v[3][2]={{-0.5, 0.0},
                 {0.5, 0.0},
                 {0.0, 0.866}};

int n; /* número de subdivisões */
```

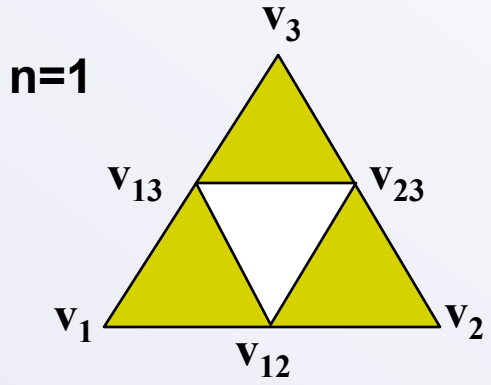
Desenha um triângulo

```
void triangulo(GLfloat *a, GLfloat *b, GLfloat *c)
{
   /* desenha um triângulo */
   glVertex2fv(a);
   glVertex2fv(b);
   glVertex2fv(c);
}
```

display(), init()

```
void display()
{
     glClear(GL_COLOR_BUFFER_BIT);
     glBegin(GL_TRIANGLES);
        divide_triangulo(v[0],v[1],v[2],n);
     glEnd();
     glFlush();
}

void init()
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-1.0, 1.0, -1.0, 1.0);
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glColor3f(0.0,0.0,0.0);
}
```

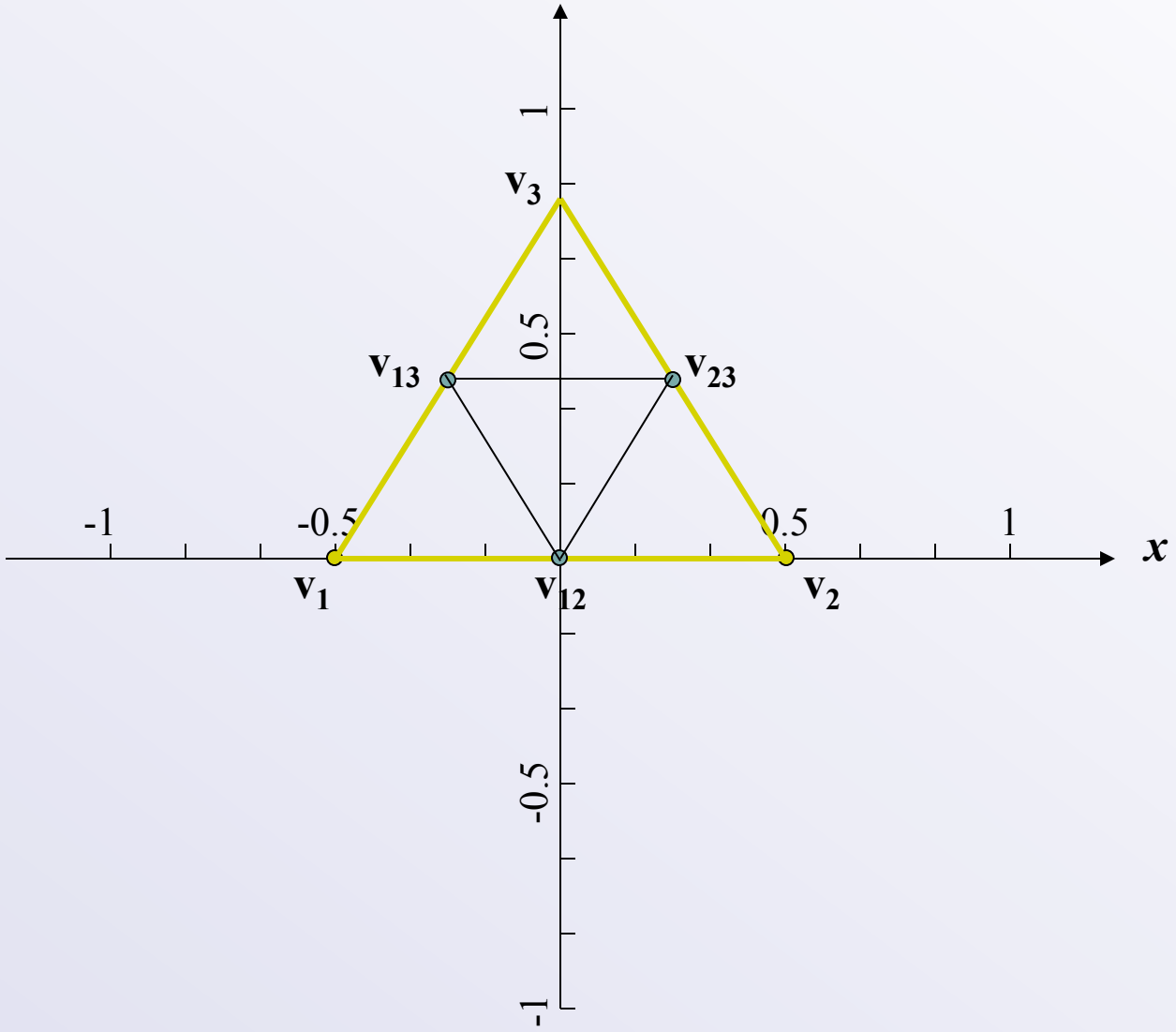




main()

```
int main(int argc, char **argv)
{
    n=4;
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutCreateWindow(“Sierpinski 2D");
    glutDisplayFunc(display);
    init();
    glutMainLoop();
}
```

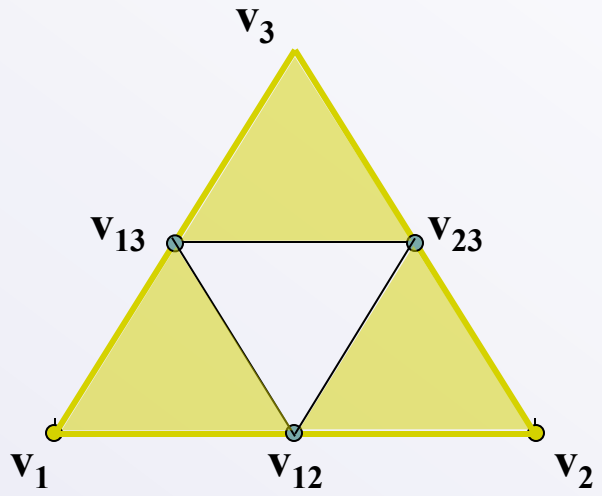
Subdivisão (n=1)



v_{12}= ?
v_{23}= ?
v_{13}= ?

Subdivisão

```
void divide_triangulo(
   GLfloat *v1, GLfloat *v2,
   GLfloat *v3, int n)
{
     if (n > 0)
     {
       /* calcular v13,v23,v12 */

       /* dividir v1, v12, v13 */
       /* dividir v12, v2, v23 */
       /* dividir v13, v23, v3 */

     /* decrementar n */
   } else triangulo(v1, v2, v3);
}
```



Subdivisão

```
void divide_triangulo(GLfloat *v1, GLfloat *v2,
                      GLfloat *v3, int n)
{
    GLfloat v12[2], v23[2], v13[2];
    if (n > 0)
    {
            for(int j=0; j<2; j++) v12[j]=(v1[j]+v2[j])/2;
            for(int j=0; j<2; j++) v23[j]=(v2[j]+v3[j])/2;
            for(int j=0; j<2; j++) v13[j]=(v1[j]+v3[j])/2;
            divide_triangulo(v1, v12, v13, n-1);
            divide_triangulo(v12, v2, v23, n-1);
            divide_triangulo(v13, v23, v3, n-1);
      }
    /* ao final da recursão, desenha triângulo */
    else triangulo(v1, v2, v3);
}
```

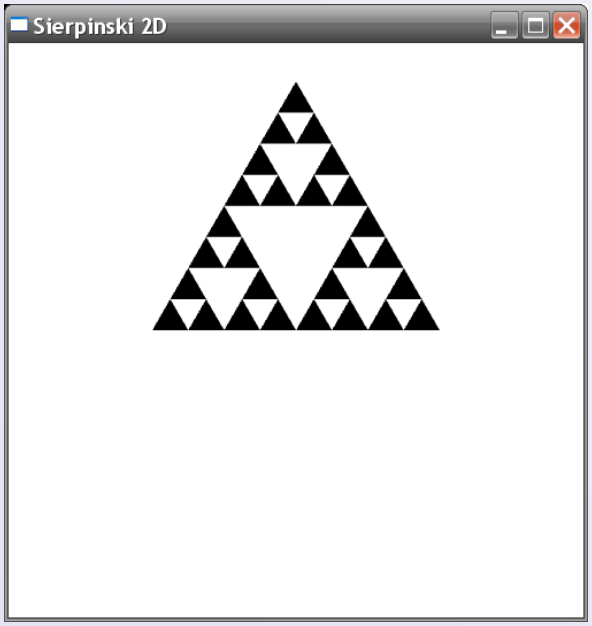
Demo Sierpinski2D.c

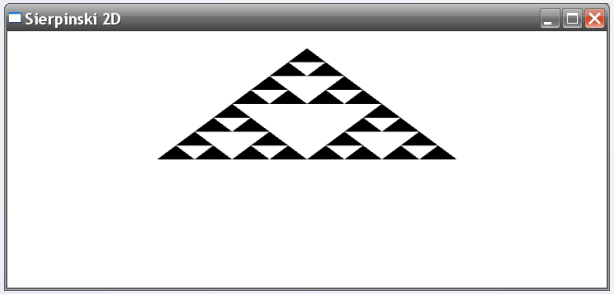
Triângulos ou polígonos?

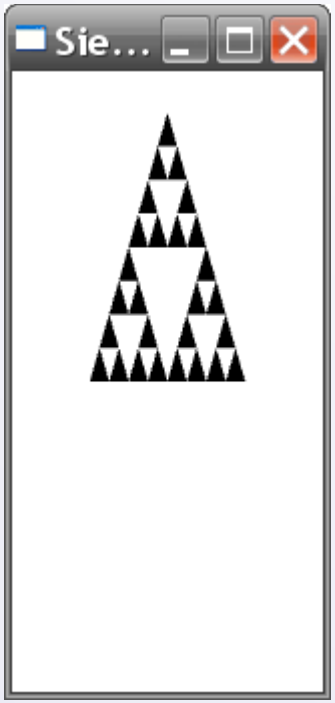
```
glBegin(GL_TRIANGLES);
   glVertex2fv(v[0]);  // v1
   glVertex2fv(v[1]);  // v2
   glVertex2fv(v[2]);  // v3
  …
glEnd();

  ou ?

glBegin(GL_POLYGON)
   glVertex2fv(v[0]);  // v1
   glVertex2fv(v[1]);  // v2
   glVertex2fv(v[2]);  // v3
  …
glEnd();
```

Sierpinski 2D

- Relação de aspecto da projeção não é preservada

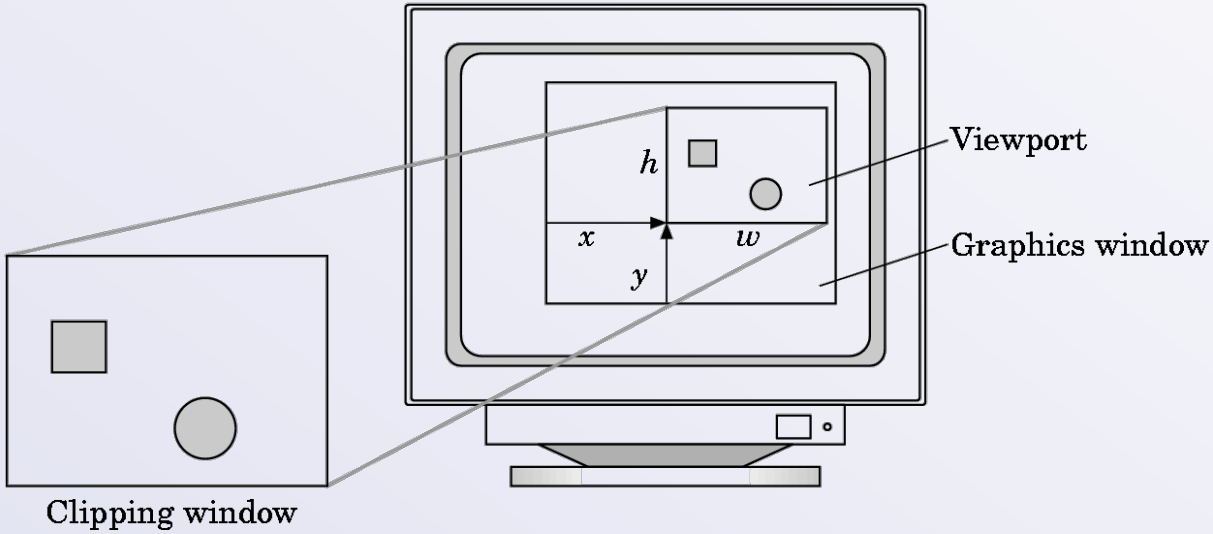






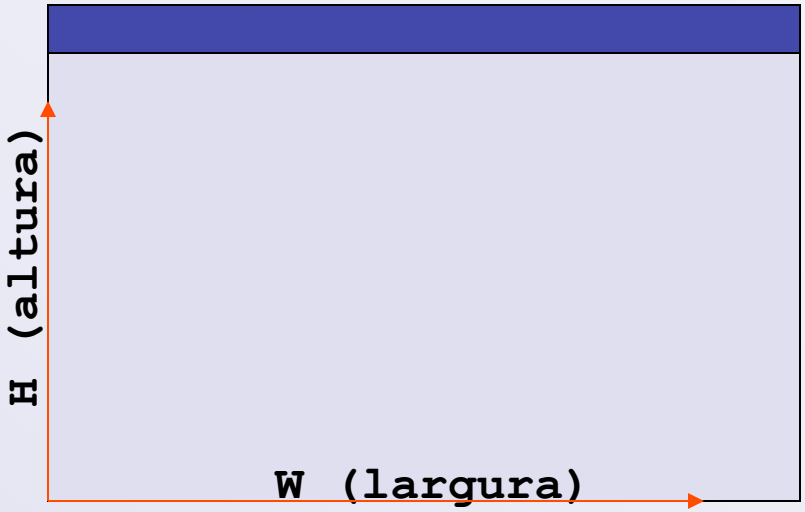
Viewports

- Não precisamos utilizar toda a janela para a visualização: **`glViewport(x,y,w,h)`**
- Valores em pixels



Callback de reshape

- `glutReshapeFunc(void (*func)(int w, int h))` indica qual ação a ser tomada quando a janela é redimensionada.



Correção da relação de aspecto

```
void reshape(int w, int h)
{
   int x = 0, y = 0;
   if (w > h) {
         x = (w - h) / 2;
         w = h;
   } else if (w < h) {
         y = (h - w) / 2;
         h = w;
   }

   glViewport((GLint)x, (GLint)y, (GLint)w, (GLint)h);
}

int main(int argc, char **argv)
{
   n=4;
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(500, 500);
   glutCreateWindow("Sierpinski 2D");
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   init();
   glutMainLoop();
}
```

Demo Sierpinski2D-1.c

- Corrigindo relação de aspecto com glViewport
- Utilizando callback de redimensionamento de janela (glutReshapeFunc)

O callback de mouse

```
glutMouseFunc(mymouse)
void mouse(GLint button, GLint
  state, GLint x, GLint y)
```

- Retorna
  - Botão (**GLUT_LEFT_BUTTON**, **GLUT_MIDDLE_BUTTON**, **GLUT_RIGHT_BUTTON**)
  - Estado do botão (**GLUT_UP**, **GLUT_DOWN**)
  - Posição na janela

Sierpinski2D-2.c

- Quando o botão esquerdo for pressionado, incrementa o número de subdivisões;
- Quando o botão direito for pressionado, decrementa o número de subdivisões;
- Redesenha a figura;

Modificações

```
void mouse(int btn, int state, int x, int y)
{
   if (btn==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
         n++;
   if (btn==GLUT_RIGHT_BUTTON && state==GLUT_DOWN)
         n > 0 ? n-- : n;
    glutPostRedisplay();
}

int main(int argc, char **argv)
{
   n=0;
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(500, 500);
   glutCreateWindow("Sierpinski 2D");
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouse);
      init();
   glutMainLoop();
}
```

Demo Sierpinski2D-2.c

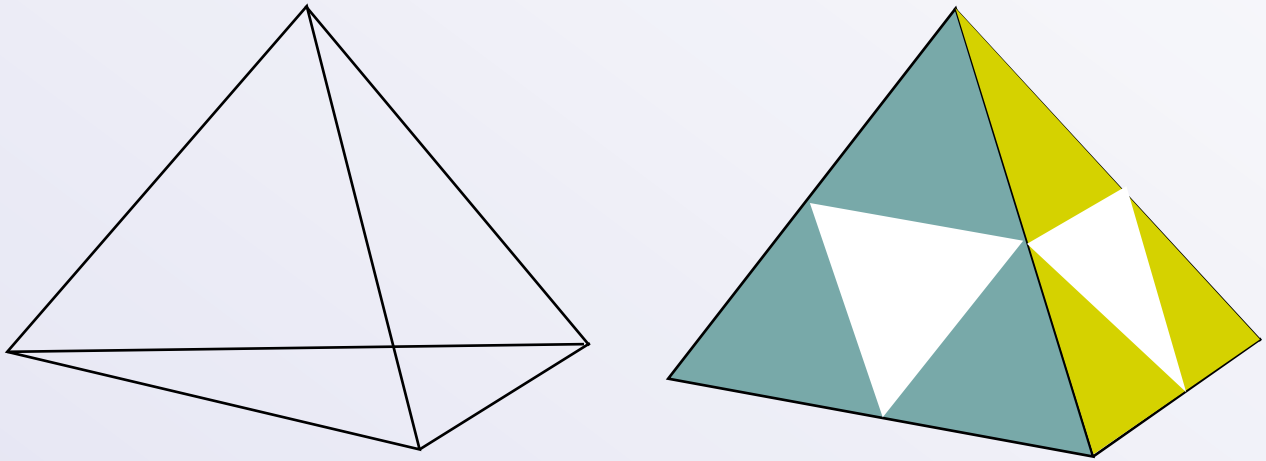
- Utilizando a callback de mouse
  `glutMouseFunc(mouse);`
- Requisitando um evento de display com
  `glutPostRedisplay();`

Aplicações em 3D

- Em OpenGL, gráficos 2D são um caso especial de gráficos em 3D
- Uso de `glVertex3*()`
- Deve-se considerar a ordem em que os polígonos serão desenhados ou usar remoção de superfícies escondidas;
- Polígonos devem ser simples, convexos e planares
- Deve-se se considerar a ordenação dos vértices (sentido horário ou anti-horário) para cada face;
- Projeções paralela (ortogonal) ou perspectiva

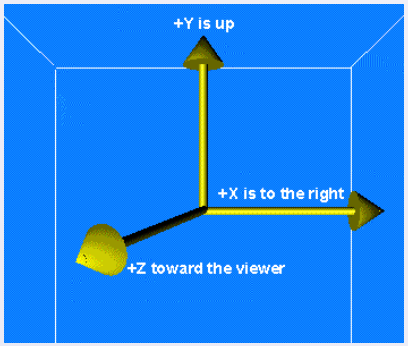
Sierpinski em 3D

- Podemos subdividir as quatro faces triangulares de um tetraedro

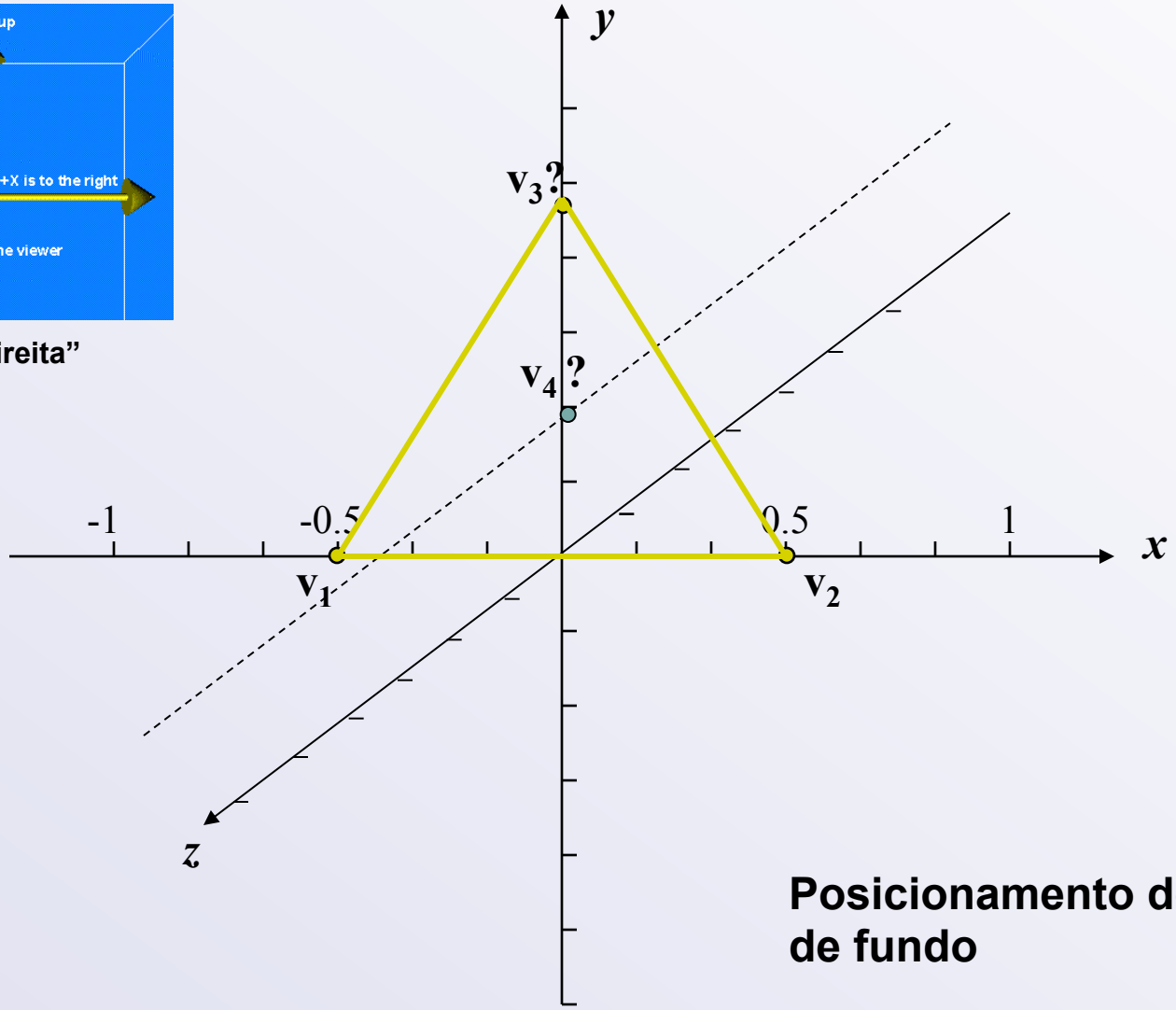


- Aparecerá como se tivéssemos removido um tetraedro do centro

Definindo o tetraedro

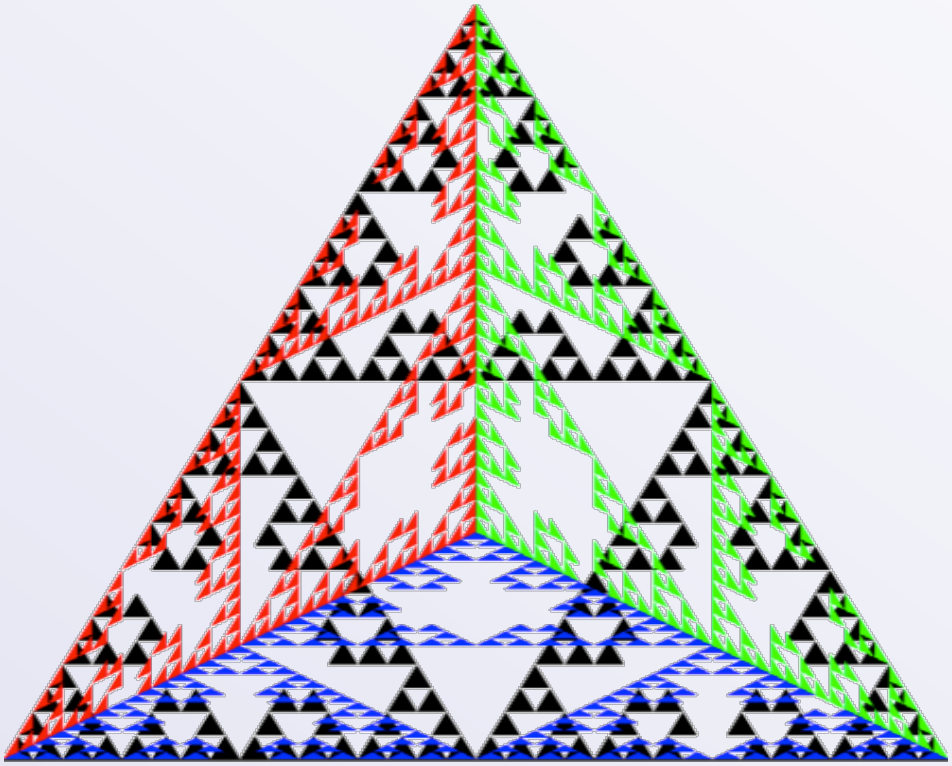


"mão direita"



Posicionamento do triângulo de fundo

Exemplo depois de 5 subdivisões

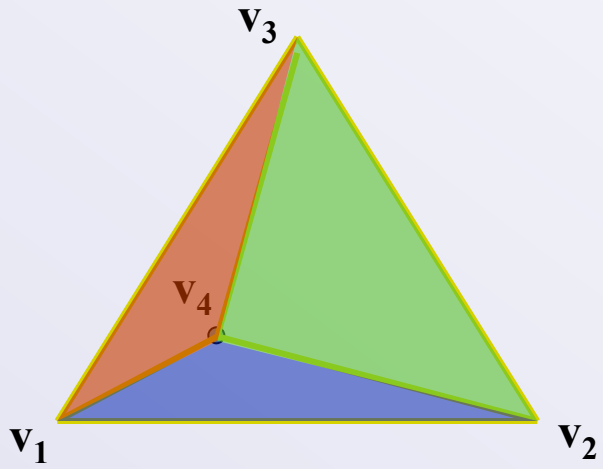


Triângulo em 3D

```
/* triângulo inicial */
GLfloat v[4][3] = {{-0.5, 0, 0},{0.5, 0, 0},
  {0, 0.866, 0.5},{0, 0.288, 0.866}};

void triangulo(GLfloat *v1,
               GLfloat *v2,
               GLfloat *v3)
{
     glVertex3fv(v1);
     glVertex3fv(v2);
     glVertex3fv(v3);
}
```

Desenhando o tetraedro

```
void tetraedro(int n)
{
    glColor3f(1.0,0.0,0.0);
    divide_triangulo(v[0], v[3], v[2], n);
    glColor3f(0.0,1.0,0.0);
    divide_triangulo(v[3], v[1], v[2], n);
    glColor3f(0.0,0.0,1.0);
    divide_triangulo(v[0], v[1], v[3], n);
    glColor3f(0.0,0.0,0.0);
    divide_triangulo(v[0], v[1], v[2], n);
}
```



Subdivisão

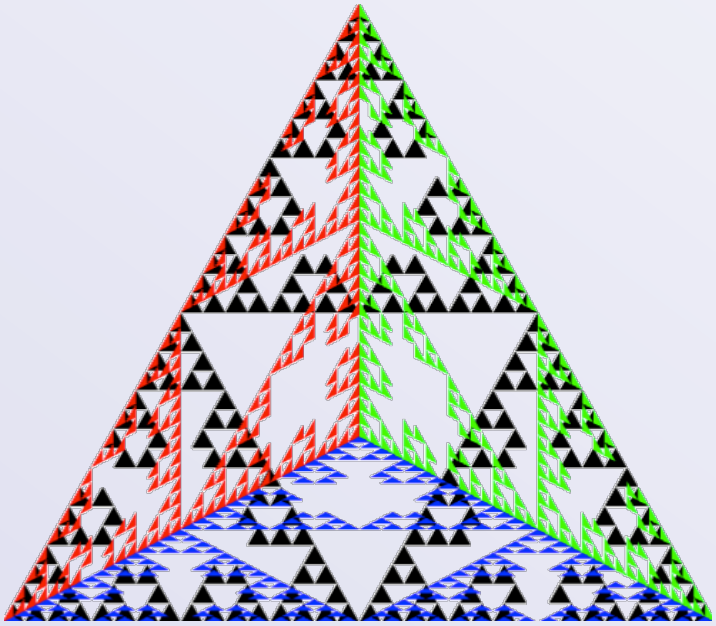
```
void divide_triangulo(GLfloat *v1, GLfloat *v2,
                      GLfloat *v3, int n)
{
    int j;
    GLfloat v12[3], v23[3], v13[3];
    if(n > 0)
    {
        for(j=0; j<3; j++) v12[j]=(v1[j]+v2[j])/2;
        for(j=0; j<3; j++) v23[j]=(v2[j]+v3[j])/2;
        for(j=0; j<3; j++) v13[j]=(v1[j]+v3[j])/2;
        divide_triangulo(v1, v12, v13, n-1);
        divide_triangulo(v12, v2, v23, n-1);
        divide_triangulo(v13, v23, v3, n-1);
    }
    /* ao final da recursão, desenha triângulo */
    else triangulo(v1, v2, v3);
}
```

Demo Sierpinski3D.c

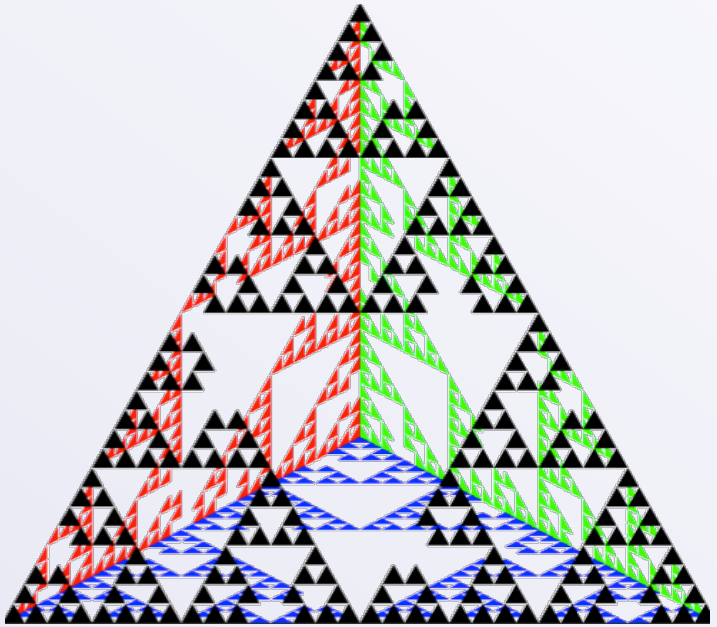
Quase lá

- Os triângulos são desenhados na ordem em que são definidos no programa. Os triângulos da frente nem sempre são renderizados na frente dos triângulos de trás.

obtido

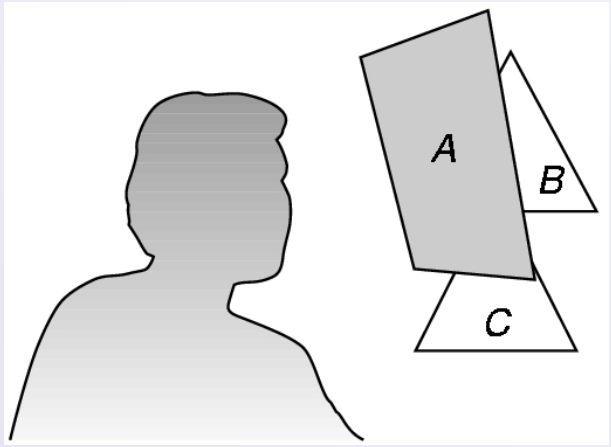


desejado



Remoção de superfícies escondidas

- Desejamos ver somente superfícies que estão realmente na frente de outras
- OpenGL utiliza o método chamado de "*z*-buffer" que salva a profundidade de cada ponto de modo que somente objetos que estão na frente são visíveis



Utilizando z-buffer

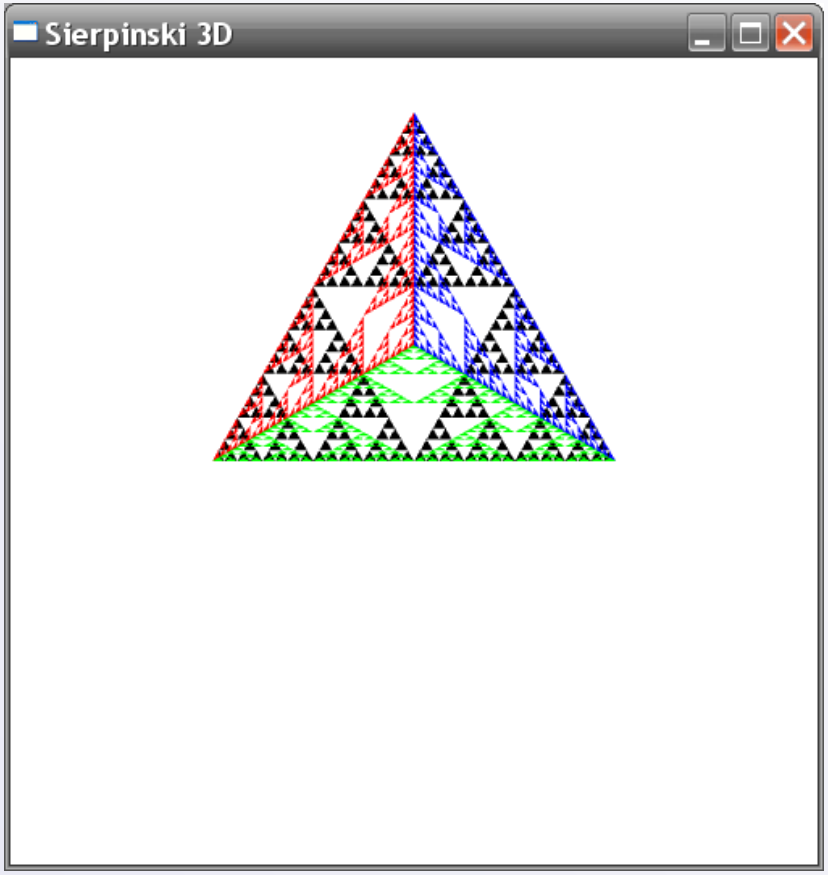
- Requisitada dentro de **`main.c`**
  - `glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH)`
- Deve ser ligada em **`init.c`**
  - `glEnable(GL_DEPTH_TEST)`
- Limpa no callback de display
  - `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`

Demo Sierpinski3D-1.c

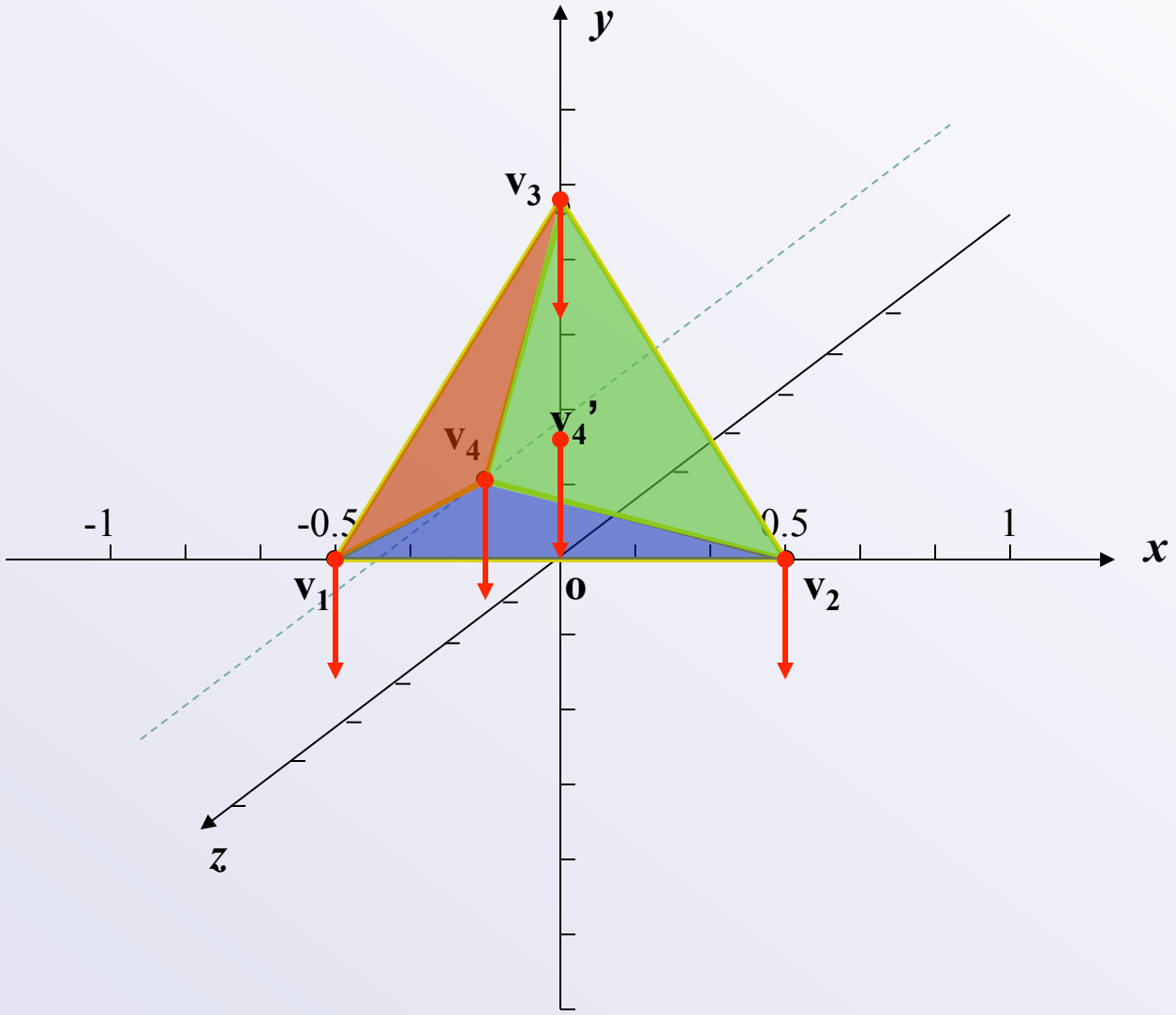
- Utilizando depth buffer para remoção de superfícies escondidas

Centralizando o tetraedro

- Tetraedro não foi desenhado com o seu centro na origem do sistemas de coordenadas do mundo real (SRU)
- Como centralizá-lo ?
  - Recalcular as coordenadas de cada vértice manualmente
  - Transformar vértices por meio de uma translação



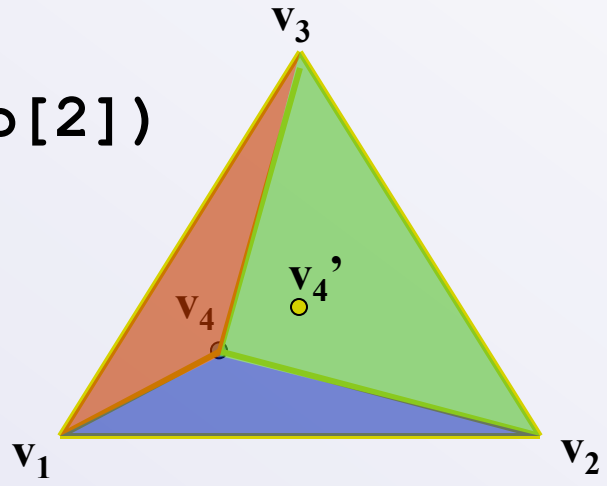
Transladando o tetraedro



Como centralizá-lo ?

- 1º. Passo: calcular baricentro de $v_1v_2v_3v_4$
  - **B** = (v_1 + v_2 + v_3+ v_4) / 4;
- 2º. Passo: deslocar cada um dos vértices por **-B**

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(-b[0],-b[1],-b[2])
```



Tetraedro centralizado

```
void init()
{
    int i;
    GLfloat b[3];

    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1, 1, -1, 1, -1, 1);

    /* calcular o baricentro */
    for (i=0; i<3; i++)
        b[i] = (v[0][i] + v[1][i] + v[2][i] + v[3][i]) / 4;

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(-b[0],-b[1],-b[2]);

    glClearColor(1.0, 1.0, 1.0, 1.0);
    glColor3f(0.0,0.0,0.0);
}
```

Demo Sierpinski3D-2.c

- Utilizando tranformações de modelo para transladar tetraedro para a origem do volume de visualização

Renderização direta

- `glBegin(GL_TRIANGLES);`
- `glVertex3f ( …);`
- …
- `glVertex3f ( …);`
- `glEnd();`

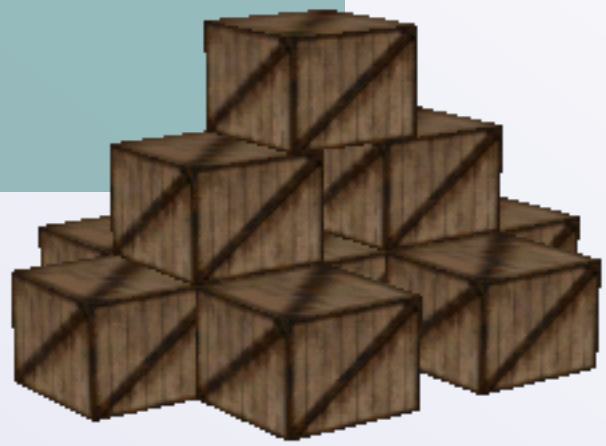
- No caso de polígonos com um número fixo de vértices, pode-se gerar vários destes polígonos usando somente um **`glBegin/glEnd`**

Display lists

- Armazena comandos do OpenGL API na memória para rápido acesso.
- Uma vez criada, não pode ser modificada.

```
static GLuint index = 0;

index = glGenLists(1)
glNewList(index, GL_COMPILE);
  // desenha algo
glEndList();
```

```
glCallList(index);
```



Vertex arrays

- Armazena vértices em arrays para reduzir o número de chamadas OpenGL

```
GLfloat vertices[] = { …};
GLfloat normals[] = {…};
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 10);
```

- Constrói triângulos usando os 10 primeiros elementos. O argumento 0 para o array indica quantos elementos devem ser pulados a cada vez.

Tutoriais de Nate Robins

- http://www.xmission.com/~nate/tutors.html
  - Projeções
  - Transformações
  - Primitivas
  - Texturas

Tarefa #3

- Criar um programa em OpenGL e GLUT que aproxime um círculo utilizando subdivisões sucessivas de um triângulo equilátero;
- Crie um callback de mouse, de forma que quando o botão esquerdo for pressionado, o triângulo será subdividido;
- MAC5744: Faça a mesma tarefa em 3D, mas de forma a aproximar uma esfera utilizando como base um tetraedro.