Universidade do Estado do Amazonas (UEA)
Escola Superior de Tecnologia (EST)
Curso: Engenharia de Computação
Data: 4 de dezembro de 2019
Disciplina: Computação Gráfica
Professor: Ricardo Barbosa
Alunos:
Carlos Diego Ferreira de Almeida
Jean Phelipe de Oliveira Lima
João Victor Melo de Oliveira
Luiz Carlos Silva de Araújo Filho

Atividade Prática
Transformações Lineares com OpenGL

Uma transformação linear é uma função que leva vetores de um espaço vetorial para outro espaço vetorial. Essa aplicação é alvo de estudo da Álgebra Linear. *OpenGL* é uma API (Interface de Programação de Aplicação) gráfica que facilita cálculos de transformações lineares e, consequentemente, o desenvolvimento de projetos em computação gráfica. Este relatório tem por objetivo descrever a solução dos autores para realização das transformações lineares utilizando *OpenGL*.

1 Fundamentação Teórica

Nesta seção são descritos os principais conceitos e técnicas que serviram de base para o desenvolvimento deste trabalho.

1.1 Conceitos de Álgebra Linear

Computação gráfica tem suas bases na álgebra linear. Ela fornece as ferramentas matemáticas necessárias para que se torne possível a apresentação de objetos (e modificações sobre os mesmos) numa tela de computador.

1.1.1 Plano Cartesiano

É um sistema de coordenadas inventando por René Descartes muito utilizado na matemática, pois é uma maneira de se representar graficamente expressões algébricas.

O plano, ilustrado na Figura 19, é composto por duas retas perpendiculares e orientadas, uma horizontal e outra vertical. À reta horizontal, dá-se o nome de **eixo x** ou **eixo das abscissas**; à reta vertical, **eixo y** ou **eixo das ordenadas**. A orientação positiva das retas é representada por uma seta.

No plano cartesiano, pares de coordenadas do tipo **(x, y)** são representados como pontos num plano 2D. Um conjunto de pontos traçados podem representar das mais simples até as mais complexas das funções algébricas existentes.

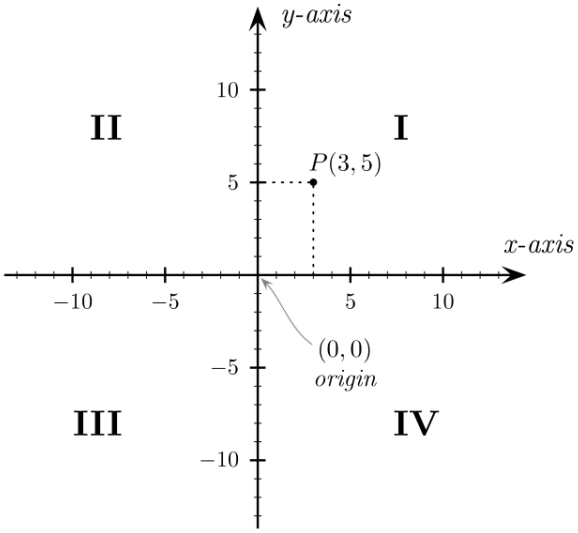

Figura 1: Plano cartesiano

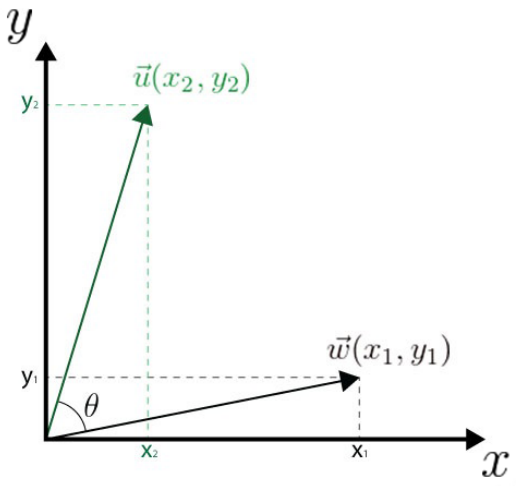

Figura 2: Vetores no plano

1.1.2 Coordenadas Homogêneas

A subseção anterior introduziu a ideia de coordenadas no plano cartesiano 2D. Em computação gráfica, há uma impossibilidade de realizar certas transformações em objetos gráficos a partir de matrizes 2 x 2. Para contornar esse empecilho, coordenadas homogêneas são utilizadas.

Uma coordenada homogênea é um triplo (x, y, t) de números reais equivalente a $(x/t, y/t)$ no plano 2D convencional. Elas são utilizadas em geometria projetiva pois facilitam uma série de operações algébricas e possibilitam representar pontos e direções que tendem ao infinito. Com esse sistema, é possível regularizar as transformações geométricas necessárias em computação gráfica.

1.1.3 Vetores

Um vetor geométrico no plano cartesiano é uma classe de objetos matemáticos (segmentos) com mesma direção, sentido e módulo. Um vetor, ilustrado na Figura 20, é formado a partir de 2 pares ordenados no plano, conhecidos como ponto de origem e de ponto extremidade. As coordenadas do vetor são obtidas pela diferença entre esses pontos.

As características de um vetor são definidas como a seguir:

1. A direção é dada pela reta que define o segmento;
2. O sentido é dado pelo sentido do movimento;
3. O módulo é o comprimento do segmento.

1.1.4 Matrizes

Uma matriz é um arranjo de números, símbolos, letras, etc., dispostos em linhas e colunas. Uma matriz com **m** linhas e **n** colunas é uma matriz de ordem **m x n**. Por

convenção, matrizes são representadas por letras maiúsculas enquanto seus elementos o são por letras minúsculas. Abaixo, X é uma matriz $\mathbf{m \times n}$ genérica e A, uma $\mathbf{3 \times 4}$

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{d1} & x_{d2} & x_{d3} & \dots & x_{dn} \end{bmatrix} \quad A = \begin{bmatrix} 5 & 2 & 3 & 11 \\ 21 & 10 & 7 & 8 \\ 15 & 9 & 6 & 16 \end{bmatrix}$$

1.1.5 Produto de Matrizes

É possível realizar diversas operações entre matrizes como adição, subtração, produto por escalar dentre outras. Essas operações carregam diversas propriedades consigo.

A operação descrita nessa seção é a multiplicação de matrizes, a mais importante das operações para computação gráfica.

Dadas as matrizes $A = [a_{ik}]_{m\,x\,t}$ e $B = [b_{kj}]_{t\,x\,n}$, o produto das matrizes A e B é uma matriz $C = [c_{ij}]_{m\,x\,n}$ cujos elementos c_{ij} são da forma:

$$c_{ij} = \sum_{k=1}^{t} a_{ik} b_{kj}.$$

O número de colunas da primeira matriz *deve* ser igual ao número de linhas da segunda. Além, o produto de matrizes não é comutativo, isto é, a ordem na multiplicação pode alterar a matriz resultante.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

Se multiplicarmos as matrizes A e B acima, a matriz C resultante será igual a matriz mostrada abaixo.

$$C = \begin{bmatrix} c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & c_{12} = a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & c_{13} = a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ c_{21} = a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & c_{22} = a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & c_{23} = a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ c_{31} = a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & c_{32} = a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & c_{33} = a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{bmatrix}$$

1.1.6 Transformações Lineares

Uma transformação linear (TL) é uma aplicação que leva vetores de um espaço vetorial em outro espaço vetorial. Representa-se uma transformação linear como $T : V \to W$, onde T é a *transformação linear* de V em W, V (um espaço vetorial) é o domínio e W (outro espaço vetorial) é o contradomínio. Para $\mathbf{x} \in \mathbb{R}^m$, o vetor $T(\mathbf{x}) \in \mathbb{R}^m$ é chamado de imagem de $\mathbf{x}$ (sob a ação de T).

É importante ressaltar que uma TL satisfaz as seguintes condições:

1. Quaisquer que sejam $\mathbf{u}$ e $\mathbf{v}$ em V, $T(\mathbf{u} + \mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v})$;
2. Quaisquer que sejam λ em $\mathbb{R}$ e $\mathbf{u}$ em V, $T(\lambda\mathbf{u}) = \lambda T(\mathbf{u})$;
3. 1. e 2. podem ser resumidas em $\forall\, \lambda \in \mathbb{R}$, $T(\lambda\mathbf{u} + \mathbf{v}) = \lambda T(\mathbf{u}) + T(\mathbf{v})$.

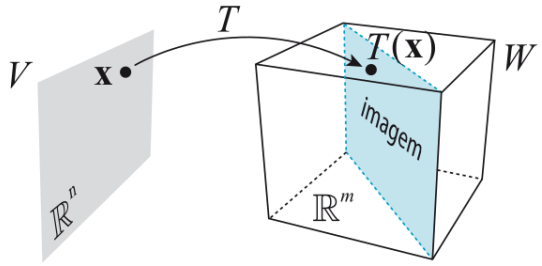

Figura 3: Domínio, contradomínio e imagem de $T : \mathbb{R}^n \to \mathbb{R}^m$

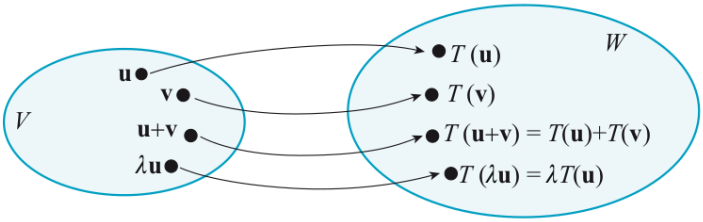

Figura 4: Definição de um TL

1.1.7 Transformações Geométricas

Transformações geométricas são, essencialmente, o mesmo que transformações lineares. Decidimos fazer essa distinção de nomenclatura por questões de clareza e entendimento. As mais básicas dessas transformações são translação, rotação e variação de tamanho (*scaling*).

Transformações geométricas em computação gráfica são de grande importância, pois com elas somos capazes de realizar operações de posicionamento, modelagem e visualização de objetos tanto em 2D quanto em 3D.

1.1.8 Translação 2D

Transladar um ponto $(\mathbf{x}, \mathbf{y})$ significa deslocá-lo no plano para alguma direção de acordo com um valor $(\Delta\mathbf{x}, \Delta\mathbf{y})$ desejado.

Forma matricial em coordenadas homogêneas:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

1.1.9 Rotação 2D

Rotacionar um ponto $(\mathbf{x}, \mathbf{y})$ em um ângulo θ relativo à origem significa encontrar outro ponto $(\mathbf{x}', \mathbf{y}')$ sobre uma circunferência centrada na origem que passa pelos dois pontos, onde $\theta = \angle$ POQ.

Forma matricial em coordenadas homogêneas:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

1.1.10 Variação de tamanho 2D

Variar o tamanho de um objeto gráfico é multiplicar cada um de seus pares ordenados $(\mathbf{x}, \mathbf{y})$ por um escalar $\lambda \in \mathbb{R}$ qualquer. Essa transformação é também conhecida como escala ou *scaling*.

Forma matricial em coordenadas homogêneas:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \lambda_x & 0 & 0 \\ 0 & \lambda_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

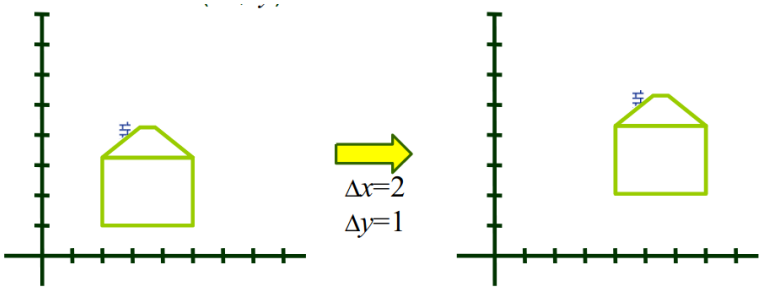

Figura 5: Translação: representação gráfica.

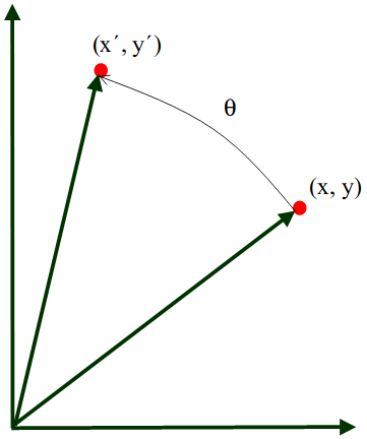

Figura 6: Rotação: representação gráfica.

1.1.11 Reflexão 2D

Essa transformação geométrica geralmente é realizada ao se compor as três transformações anteriores em uma multiplicação encadeada de matrizes na ordem translação, rotação, escala. Porém, em essência, reflexão nada mais é que uma escala com valores de coordenadas simétricos. Nesse caso, o objeto é refletido sobre seu próprio eixo.

Quando se deseja refletir um objeto sobre os eixos de um plano, os elementos da matriz de reflexão devem ser os eixos $\mathbf{x}, \mathbf{y}, \mathbf{z}$ do plano.

Forma matricial em coordenadas homogêneas (repare que a matriz é a mesma da *scaling*, com valores simétricos nos escalares $\mathbf{x}, \mathbf{y}$):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -\lambda_x & 0 & 0 \\ 0 & -\lambda_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

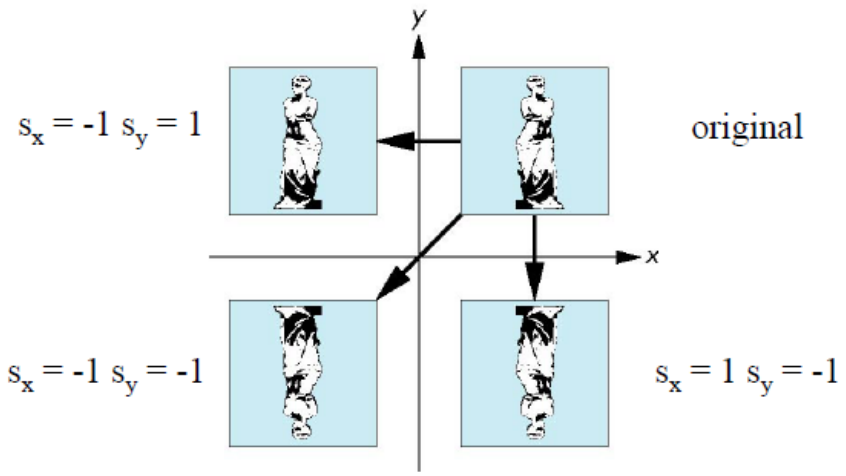

Figura 7: Reflexão sobre os eixos do plano.

1.2 Conceitos de OpenGL

OpenGL é uma das APIs gráficas mais utilizadas na atualidade. Ela serve de *middleware*, facilitando cálculos vetoriais e matriciais necessários à computação gráfica. Uma outra API associada ao *OpenGL* é a GLUT, que facilita a criação de janelas e afins.

Algumas características do *OpenGL* são:

(a) É uma máquina de estados. Algumas funções do *OpenGL* guardam flags necessárias às configurações do programa como as características de cor de fundo de tela, configurações de janelas e outras habilitações ao programa;

(b) É orientada a eventos. O programa escrito utilizando a OpenGl permanece em loop até que algum evento aconteça. Na especificação dos eventos está mencionada a função *callback* que será chamada toda a vez que o evento a ela relacionado acontecer. O programa só se encerra se o evento correspondente ao final da execução for ativado.

(c) É independente da plataforma. As funções da *OpenGL* são padronizadas quanto ao formato passado ao usuário e ao programador. As diferenças em relação ao tratamento do sistema de janelas e do *hardware* existem apenas internamente ao código da API. Dessa forma os parâmetros e as funções não mudam e um código escrito para um compilador de uma plataforma UNIX, por exemplo, compila tranquilamente em um do *Windows* e vice-versa.

1.2.1 Sistemas de Coordenadas

O sistema de coordenadas de uma aplicação *OpenGL* tem origem no canto inferior esquerdo da tela (em relação ao observador). Assim os valores de **y** crescem à medida que o ponto se aproxima do topo da tela, e os valores de **x** crescem à medida que o ponto se aproxima da lateral direita da tela.

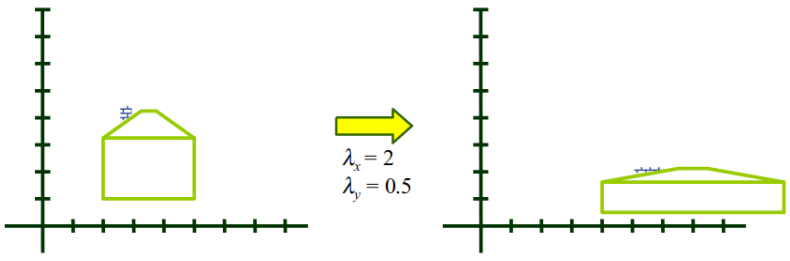

Figura 8: Scaling: representação gráfica.

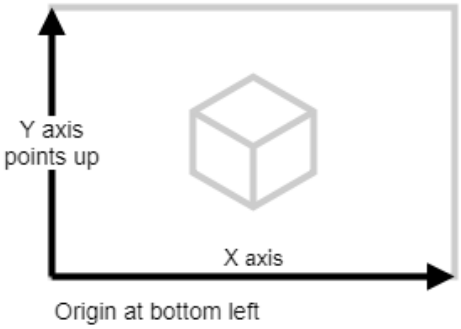

Figura 9: sistema de coordenadas do *OpenGL*.

1.2.2 Inicializando uma janela

Para desenhar utilizando um programa *OpenGL* é necessária uma janela de visualização. As configurações da janela ou das janelas de visualização são feitas na função `main()`.

O código a seguir inicializa uma janela vazia de acordo com as configurações dadas:

```
void main (int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(640, 480);
   glutInitWindowPosition(100, 150);
   glutCreateWindow(argv[0]);
   glutDisplayFunc(desenho);
   glutMainLoop();
}
```

`glutInit(&argc, argv)` Esta função inicializa a GLUT. Os argumentos poderão ser passados como parâmetros para o programa, como foi feito com o parâmetro `argv`, que representa o nome da janela, conforme especificado pela função `glutCreateWindow()`.

`glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)` Essa função especifica como o desenho será inicializado. A constante `GLUT_SINGLE` indica que será utilizado apenas um *buffer* para alocar a imagem – ou seja, o desenho será construído diretamente na tela –, e diz que a imagem será feita utilizando o padrão de cores RGB para colorir.

`glutInitWindowSize(640, 480)` Esta função especifica qual o tamanho inicial da janela. No caso são 640 *pixels* de largura por 480 *pixels* de altura. Quando o programa está rodando, o usuário pode redimensionar a janela, se preferir.

`glutInitWindowPosition(100, 150)` Esta função especifica que a janela posicionada a uma distância de 100 *pixels* do limite esquerdo da tela e a 150 *pixels* do limite superior da tela.

`glutCreateWindow(argv[0])` Esta função abre uma janela de visualização de acordo com as configurações especificadas nas funções acima. Esta janela terá como nome o parâmetro passado para esta função, no caso a entrada de dados na chamada do executável de acordo com o parâmetro argv[0].

`glutDisplayFunc(desenho)` Esta é uma função *callback* necessária para a execução de qualquer programa que utilize a biblioteca GLUT (a partir da versão 3.0).

`glutMainLoop()` Esta função é responsável por manter o programa em *loop* até que seja chamado algum evento que determine o fim da aplicação. Essa função caracteriza a especificação de *OpenGL* como uma API orientada a eventos.

1.2.3 Orientação a Eventos

Como foi dito, a *OpenGL* é orientada a eventos e trata eventos de Mouse, de Teclado ou de Alterações da Janela de Visualização. A chamada dos eventos e seus *callbacks* ficam no `main()` do programa. O cabeçalho, declaração e a definição das funções são por conta do desenvolvedor.

A seguir, um trecho de código que manipula eventos a partir de funções *callback*:

```
   glutDisplayFunc(desenho);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouse);
   glutKeyboard(teclado);
   glutMainLoop();
```

glutDisplayFunc(desenho) Sempre que o programa determinar que uma janela deve ser redesenhada na tela, isso significa um evento de atualização de desenho. Neste exemplo a função `desenho()` é registrada como a função de atualização de desenho e chamada sempre que é preciso redesenhar uma cena.

glutReshapeFunc(reshape) O tamanho da janela de visualização pode ser alterado com o mouse por exemplo. Neste exemplo a função `reshape()` é registrada como um evento de atualização de tamanho de janela. A função `reshape`,passa parâmetros automaticamente para que sejam atualizados a largura e a altura da janela.

glutMouseFunc(mouse) Quando um botão do mouse é pressionado ou solto, um evento de mouse é acionado. Neste exemplo a função `mouse()` é registrada como uma função para ser chamada quando ocorrer eventos de *mouse*. A função `mouse()` recebe argumentos que identificam posição do cursor e a natureza da ação iniciadas na função.

glutKeyboardFunc(teclado) Este comando chama a função `teclado` quando uma tecla do teclado é pressionada ou solta. Os argumentos da função `teclado()` possuem argumentos que indicam que tecla foi pressionada além da posição do cursor naquele instante.

1.2.4 Desenhando Primitivas

Todos os desenhos produzidos pela *OpenGL* são gerados por primitivas geométricas básicas. Pontos, retas, sequência de retas, polígonos e outras são combinadas de forma que geram uma ampla gama de figuras que vemos em projetos que utilizam a API.

É preciso passar como parâmetro para a função de estado `glBegin()` a primitiva desejada com que se trace o desenho. Tudo o que estiver entre o `glBegin()` e o `glEnd()` será desenhado utilizando a primitiva passada como argumento.

No trecho de código abaixo, a função `desenho` desenha tudo aquilo por ela especificado utilizando pontos, que é a primitiva passada como parâmetro para a `glBegin()`.

```
glBegin(GL_POINTS);
desenho():
glEnd();
```

As funções da *OpenGL* seguem o seguinte padrão de nomeação:

$$\underbrace{\texttt{\{biblioteca\}}}_{1}\underbrace{\texttt{\{comando básico\}}}_{2}\underbrace{\texttt{\{número argumentos\}}}_{3}\underbrace{\texttt{\{tipo argumentos\}}}_{4}$$

1. indica a biblioteca à qual a função pertence;
2. é o radical da função, a função em si;
3. é a quantidade de parâmetros passados para a função;
4. é o tipo dos argumentos passados para a função.

O exemplo a seguir esclarece como o padrão funciona:

A API *OpenGL* possui seus próprios tipos de dados para evitar problemas de compatibilidade entre plataformas.

A seguir, são apresentadas as principais primitivas da *OpenGL*:

Exemplos renderizados das primitivas apresentadas são mostrados abaixo:

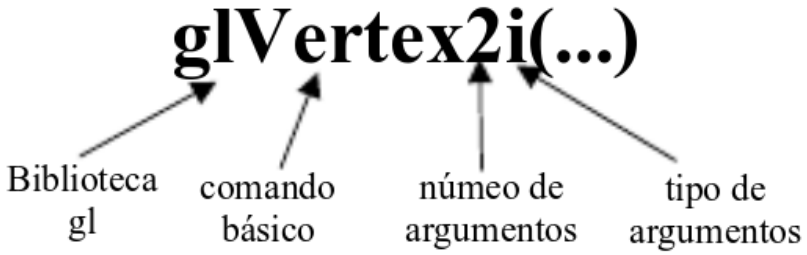

Figura 10: Padrão de nomeação: exemplo.

Sufixo	Tamanho de dados	Tipo de dado no C++	Nome na OpenGL
b	inteiro 8-bits	signed char	Glbyte
s	inteiro 16-bits	short	Glshort
i	inteiro 32-bits	int or long	Glint, Glsizei
f	floating points 32 bits	float	Glfloat, Glclampf
d	floating points 64 bits	double	Gldouble, Glclampd
ub	unsigned 8-bits	unsigned char	Glubyte, Glboolean
us	unsigned 16-bits	unsigned short	Glushert
ui	unsigned 32 bits	unsigned int or unsigned long	Gluint, enum, bitfield

Figura 11: Tipos de dados *OpenGL*

1.2.5 Transformações Geométricas em OpenGL

Assim como na álgebra se usa de transformações lineares para modificar gráficos, também na computação gráfica esses tipos de transformações são usados para dinamizar objetos. Transformações de translação, rotação e de escala do objeto obtido são as mais comuns.

Na álgebra as transformações lineares podem ser representadas por matrizes que representam a transformação geométrica desejada. Em programas que utilizam *OpenGL* são usadas matrizes para guardar as transformações realizadas sobre objetos, transformações estas que incluem ainda a translação, que não é TL porque sua aplicação sobre a origem não retorna a origem. Essas matrizes são chamadas de Matrizes de Transformação. Elas guardam a posição do objeto em determinado instante e tornam possível serem realizadas transformações sobre transformações.

No código abaixo vemos exemplo de transformações. Nele utilizamos as funções `glRotatef`, `glTranslatef` e `glScalef` que internamente são tratadas pela *OpenGL* como matrizes. No caso desse exemplo, temos um cubo que gira e se move a partir de comandos do teclado.

Primitiva	Descrição
GL_POINTS	Desenha pontos independentes
GL_LINE	Desenha uma linha entre dois pontos passados
GL_LINE_STRIP	Desenha uma sequência de linhas
GL_LINE_LOOP	Desenha uma sequência de linhas que retornam à origem
GL_QUADS	Desenha quadriláteros independentes a cada quatro pontos
GL_QUAD_STRIP	Desenha uma sequência de quadriláteros ligados
GL_TRIANGLES	Desenha triângulos independentes a cada 3 pontos
GL_TRIANGLE_STRIPS	Desenha triângulos conectados com origens diferentes
GL_TRIANGLE_FAN	Desenha triângulos conectados com origem comum
GL_POLYGON	Desenha um polígono convexo

Tabela 1: Primitivas da *OpenGL*

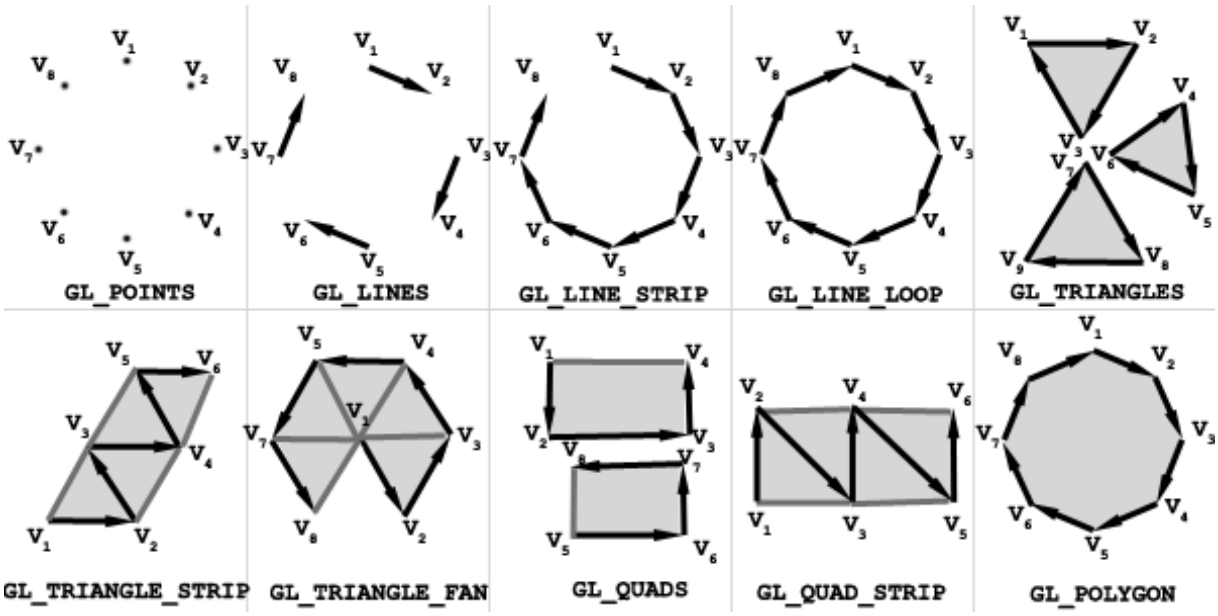

Figura 12: Exemplos de primitivas renderizadas

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

//<<<<<<<<<<<<<<<<<< Variveis Globais >>>>>>>>>>>>>>>>>>
int rotaciona = 1, translada = 0, escala = 0, largura, altura;
float angle = 0, t_x = 0, t_y = 0, e_x = 1, e_y = 1;

//<<<<<<<<<<<<<<<<<<<< Prottipos >>>>>>>>>>>>>>>>>>>>>>
void teclado(int key, int x, int y);
void reshape(int largura, int altura);
void desenho(void);
void init(void);
```

```
//<<<<<<<<<<<<<<<<<<< Main >>>>>>>>>>>>>>>>>>>>>>>>
int main (int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(300, 300);
   glutInitWindowPosition(10, 10);
   glutCreateWindow("Transformacoes");
   glutDisplayFunc(desenho);
   glutKeyboardFunc(teclado);
   glutReshapeFunc(reshape);
   init();
   glutMainLoop();
   return 0;
}

void init(void)
{
   glClearColor(1.0,1.0,1.0,0.0);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(-100, 100, -50, 50);
}

//<<<<<<<<<<<<<<<<<<< Desenho >>>>>>>>>>>>>>>>>>>>>>>>
void desenho(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();

   glTranslatef(t_x, t_y,0);
   glRotatef(angle, 0, 0, 1);
   glScalef(e_x, e_y, 1);

   glBegin(GL_POLYGON);
      glColor3f(0,0,1);
      glVertex2i(0, 0);
      glColor3f(0,1,0);
      glVertex2i(0, 20);
      glColor3f(1,0,0);
      glVertex2i(30, 10);
   glEnd();

   glFlush();
}
```

```
//<<<<<<<<<<<<<<<<<<<< Teclado >>>>>>>>>>>>>>>>>>>>>>>
void teclado(unsigned char key, int x, int y)
{
   if (translada){
      switch (key){
         case '+':
            t_x+=10;
            break;
         case '-':
            t_x-=10;
         break;
      }
   }

   if (rotaciona){
      switch (key){
         case '+':
            angle+=10;
            break;
         case '-':
            angle-=10;
            break;
      }
   }

   if (escala){
      switch (key){
         case '+':
         e_x+=1;
         break;
      case '-':
         e_x-=1;
         break;
      }
   }

   switch (key){
      case 't'|'T':
         translada = !translada;
         break;
      case 'r'|'R':
         rotaciona = !rotaciona;
         break;
      case 'e'|'E':
         escala = !escala;
         break;
   }
```

```
    printf("Translada = %c\n", translada? 'V':'F');
    printf("Rotaciona = %c\n", rotaciona? 'V':'F');
    printf("Escala = %c\n", escala? 'V':'F');
    printf("X = %f\n", t_x);
    printf("Angulo = %f\n", angle);
    printf("Escala = %f\n\n", e_x);

    glutPostRedisplay();
}

//<<<<<<<<<<<<<<<<<<< Reshape >>>>>>>>>>>>>>>>>>>
void reshape (int largura, int altura)
{
    glViewport(0, 0, largura, altura);
}
```

O exemplo anterior introduziu uma série de novas funções. Discutimo-as a seguir.

glTranslate(t_x, t_y, t_z) Para transladar um sistema é preciso multiplicar todos os pontos do sistema de coordenadas por uma matriz de translação. A função `gLTranslate` cria uma matriz de translação a partir dos parâmetros `t_x, t_y, t_z`, que serão os valores a serem somados às respectivas coordenadas x, y e z.

glRotate(angle, r_x, r_y, r_z) Para rotacionar um sistema de coordenadas é preciso multiplicar as coordenadas atuais por uma matriz de rotação. A função `gLRotate` cria uma matriz de rotação a partir dos parâmetros `angle` (que indica o ângulo de rotação), `r_x, r_y` e `r_z` (que descrevem as coordenadas de um vetor que definirá o eixo de rotação). No exemplo trabalhado, os valores das coordenadas do eixo de rotação foram (0, 0, 1), o que significa que o sistema vai girar em torno do eixo z, que é o eixo perpendicular ao plano da tela.

glScale(e_x, e_y, e_z) Para escalonar um sistema de coordenadas é preciso multiplicar cada coordenada de um objeto por um fator de escala. Através dos parâmetros `e_x, e_y, e_z`, que representam os fatores de escala relativos a cada eixo coordenado, a função `gLScale` realiza essa transformação.

glLoadIdentity() A função `gLLoadIdentity` carrega a matriz identidade na matriz de transformações no *pipeline* do *OpenGL*.

glMatrixMode() Todas as transformações acima são feitas sobre o objeto. É importante especificar que as transformações podem ser sobre os objetos e sobre a câmera. O que vai determinar que tipo de transformação será realizada é a função `gLMatrixMode`. Para que as transformações sejam realizadas sobre o objeto (sistema de coordenadas) o parâmetro dessa função deve ser `GL_MODELVIEW`.

2 Metodologia

Primeiramente, cada atividade foi dividia por integrante. Os programas foram desenvolvidos em máquinas relativamente modernas dos próprios alunos, em ambiente *Windows* ou *iOS*, e após instalação de pacotes necessários e solução de problemas do processo de instalação. Em seguida, cada um estudou individualmente uma ou mais maneiras de resolver o problema. Majoritariamente, o procedimento de desenvolvimento consistiu em

leitura de artigos pela *web* e visualização de vídeo-aulas em plataformas *online*. Após solucionar o problema, a equipe se reuniu e discutiu os códigos-fontes.

Os códigos da equipe foram desenvolvidos metade na linguagem de programação C (translação e reflexão) e metade em Python (escala e rotação) com intuito ampliar o conhecimento dos integrantes e também por questões de comodidade. Apesar da existência de rotinas prontas para transformações geométricas na *OpenGL*, os integrantes desenvolveram suas próprias rotinas de manipulação de matrizes para cada fim, conforme solicitado pelo professor. Os conceitos de álgebra linear foram aplicados aliados à criação de objetos e janelas da *OpenGL* para finalização do exercício.

3 Resultados

Nesta seção, são descritos os códigos que realizam as transformações geométricas em objetos gráficos.

3.1 Descrição Translação

A operação de translação de figuras geométricas 2D foi realizada utilizando a linguagem de programação Python em sua versão 3.6 em conjunto com as bibliotecas `OpenCV` e `NumPy` para facilitar a reprodução gráfica das matrizes. Quando o código é executado, é solicitado ao usuário que informe qual figura deve ser desenhada na tela: '`c`' para um círculo, '`t`' para um triângulo, ou '`r`' para um retângulo. A figura é desenhada pela primeira vez centralizada. O desenho da figura é feito a partir de uma matriz de 512×512 *pixels* que estão, inicialmente, com valor nas 3 dimensões igual a 0, sendo assim, pretos e, ao receber o comando de desenho da figura, muda a cor dos *pixels* determinados pelo tamanho da figura (variável alterável apenas dentro do código) para branco.

Para realizar a translação, espera-se pela entrada do usuário mostrando a direção para a qual a forma deve ser deslocada, sendo '`w`' para cima, '`s`' para baixo, '`d`' para direita e '`a`' para esquerda (a figura é sempre deslocada 5 *pixels* em qualquer uma das direções). Em seguida multiplica-se a matriz de translação por uma matriz coluna composta pelos pontos **x** e **y** de origem da forma e 1. Ao fim de cada *loop*, a tela é redesenhada, a qual recebe as formas em suas novas posições, aspecto refletido na imagem desenhada na tela. Para terminar a execução do código basta digitar '`q`'.

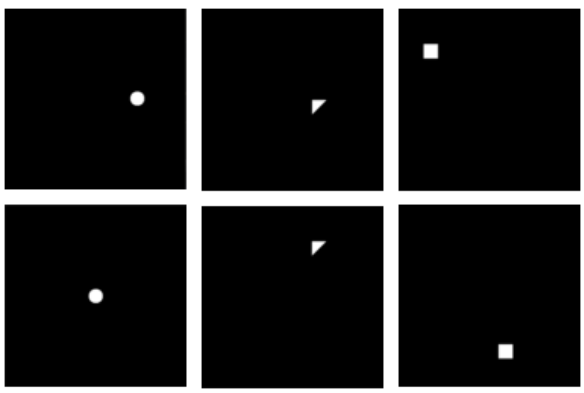

Figura 13: Translação: capturas de tela.

3.2 Descrição Rotação

A operação de rotação foi implementada utilizando a linguagem de programação Python em sua versão 3.7.

3.2.1 Utilizando OpenGL

Primeiramente, uma interação com usuário é solicitada para informar qual figura deve ser gerada, a escolha é feita inserindo o número correspondente da figura:

- Quadrado
- Triângulo
- Círculo

Uma vez escolhida a figura, o programa realiza o desenho, o mesmo já começa rotacionando. É possível alterar a figura a qualquer momento, bastando apenas clicar nos números correspondentes das figuras. Também é possível parar a rotação da figura clicando no número 4.

Para desenhar as formas foi necessário apenas informar alguns pontos e utilizar a função `glVertex2fv` do *openGL*.

Para rotacionar as formas foi utilizada a função `glRotatef` do *openGL*, onde apenas precisamos informas em qual eixo a figura irá ser rotacionada.

3.2.2 Rotação Manual

Primeiramente, uma interação com usuário é solicitada para informar qual figura deve ser gerada, a escolha é feita inserindo o número correspondente da figura:

- Quadrado
- Triângulo
- Círculo

Uma vez escolhida a figura, o programa realiza o desenho, o mesmo já começa rotacionando. É possível alterar a figura a qualquer momento, bastando apenas clicar nos números correspondentes das figuras. Também é possível parar a rotação da figura clicando no número 4.

Para desenhar as formas foi necessário fazer um vetor com todos os pontos em três dimensões da figura desejada.

Para rotacionar a figura foi preciso multiplicar todos os pontos da figura por uma matriz de rotação. Para plotar as figuras foi utilizada a biblioteca *pyGames*, os pontos em 3d foram convertidos em 2D e depois plotados.

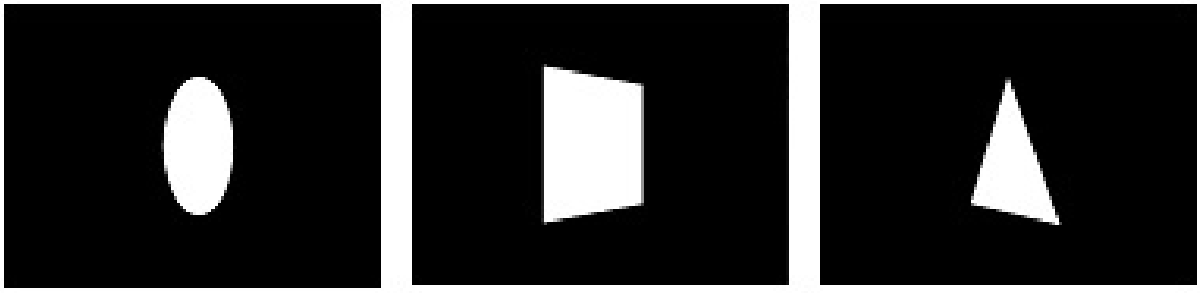

Figura 14: Rotação OpenGL: capturas de tela.

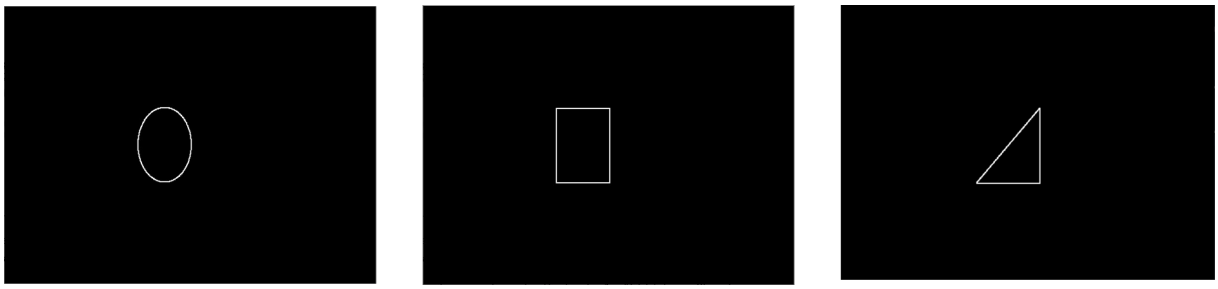

Figura 15: Rotação Manual: capturas de tela.

3.3 Descrição Escala

A operação de escala foi implementada utilizando a linguagem de programação Python em sua versão 3.7. Primeiramente, uma interação com usuário é solicitada para informar qual figura deve ser gerada: circunferência, triângulo ou quadrado. Uma vez escolhida a figura, o programa realiza o desenho, sempre centralizado. O desenho de das figuras é feito através da construção de uma matriz que contendo os vértices a serem plotados. Após todas essas construções geométricas, o usuário, por meio dos comandos de teclado '+' e '-' pode redimensionar o desenho, isto é, aplicar a transformação de escala à matriz implementada.

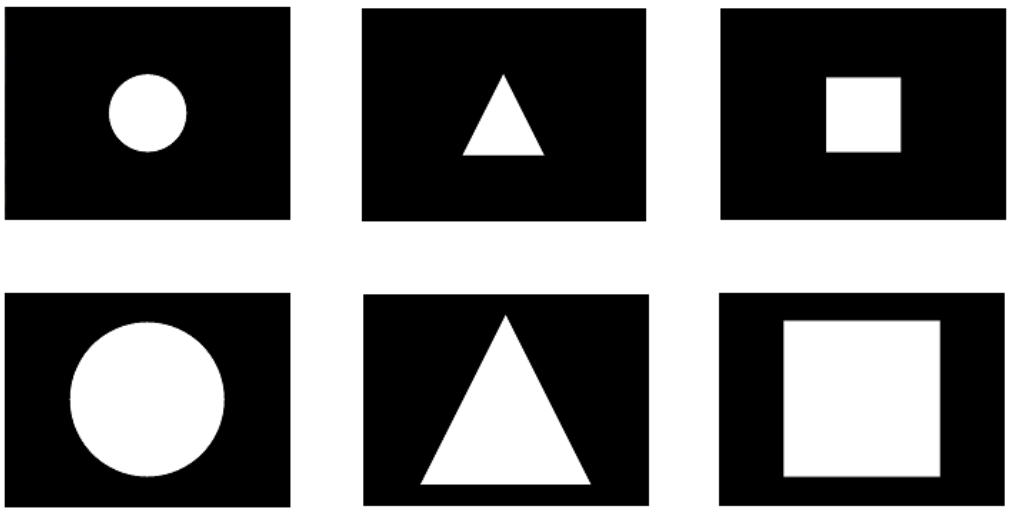

Figura 16: Escala: capturas de telas.

3.4 Descrição Reflexão

Para realização desta tarefa, foi utilizado o ambiente *Windows 10* 64-bit, a linguagem de programação C e a última versão do compilador MinGW.

O programa realiza reflexão das formas geométricas triângulo, quadrado e círculo. Ele inicia mostrando uma tela preta 600 x 600 sem conteúdo à espera de algum comando do usuário. Os possíveis comandos são enviados via teclado e são eles: **l**: limpa a tela; **t**: renderiza o triângulo; **s**: renderiza o quadrado; **c**: renderiza o círculo; $\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}$: escolhem o quadrante a qual se deseja refletir a forma.

No que diz respeito à implementação, o código segue o básico de *OpenGL* se diferenciando nas rotinas que realizam a operação de transformação linear. Ao invés de se usar as rotinas `gl` oferecidas pela API, foram criadas rotinas próprias que fazem a multiplicação da matriz que representa a forma geométrica e da matriz de transformação adequada.

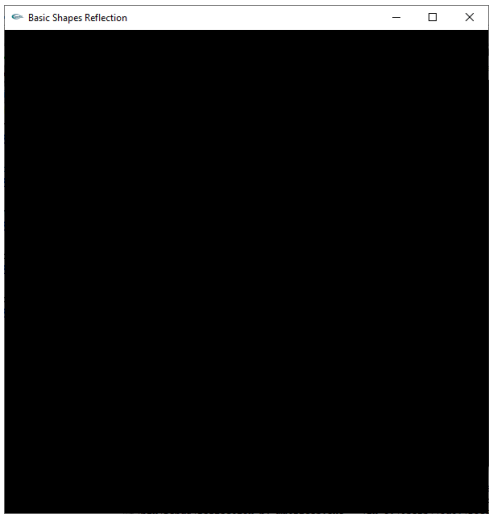

Figura 17: Reflexão: captura de tela 01. Tela inicial.

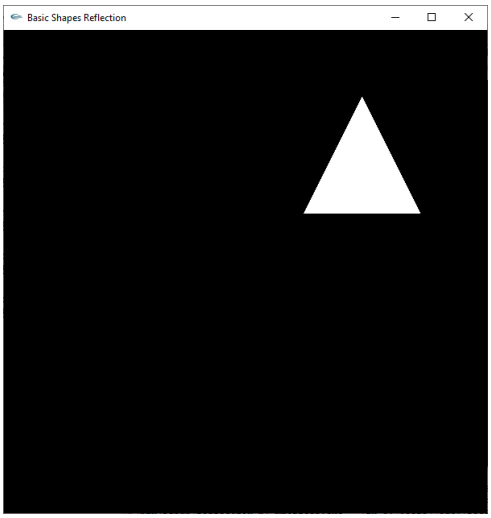

Figura 18: Reflexão: captura de tela 02. Triângulo no 2º quadrante.

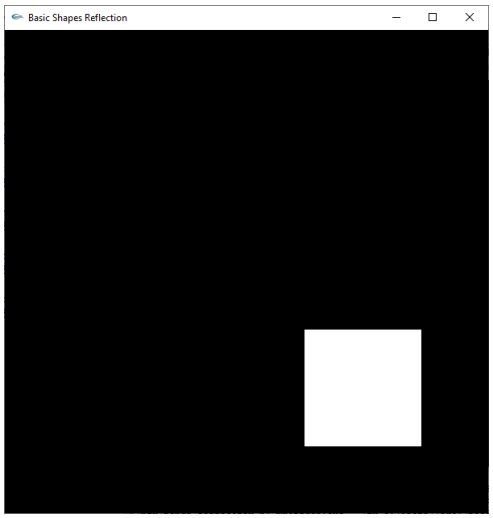

Figura 19: Reflexão: captura de tela 03. Quadrado no 4º quadrante.

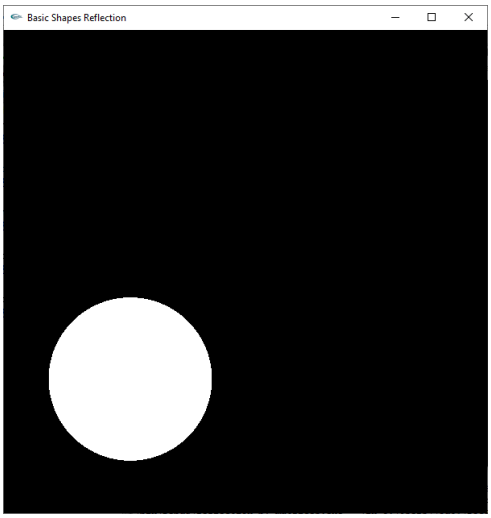

Figura 20: Reflexão: captura de tela 04. Círculo no 3º quadrante.

4 Códigos-fontes

Os códigos-fontes para cada transformação linear necessária para completar atividade podem ser encontrados nos links a seguir.

Translação: https://github.com/LuizCarlosS/PDI-e-CG/blob/master/Translacao%20manual%20python/main.py

Rotação: https://github.com/DiegoFr75/Glut/blob/master/rotate_na_mao.py
https://github.com/DiegoFr75/Glut/blob/master/rotate.py

Escala: https://github.com/jpdol/IntroGC/blob/master/scale_na_mao.py

Reflexão: https://github.com/jvmdoeng16/cg-stuffs/blob/master/basic-shapes-axis-reflection.c

Referências

[1] BEAN, S. E. P. C.; KOZAKEVICH, D. N. *Álgebra Linear I*. 2. ed. Florianópolis - SC, Brasil: FSC/EAD/CED/CFM, 2011.

[2] Fábio Franco. *Resumo sistema de coordenadas*. 2018. Disponível em `http://docente.ifrn.edu.br/fabiofranco/disciplinas/elementos-de-fisica/1a-unidade/2o-momento/resumo-sistema-de-coordenadas`. Acesso em 4 de dezembro de 2019.

[3] Ulysses Sodré. *Geometria plana e espacial - Vetores no plano cartesiano*. 2008. Disponível em `http://www.uel.br/projetos/matessencial/geometria/vetor2d/vetor2d.htm`. Acesso em 4 de dezembro de 2019.

[4] Hae Yong Kim. *Transformações geométricas em coordenadas homogêneas 2D*. 2013. Disponível em `http://www.lps.usp.br/hae/apostila/transformacao.pdf`. Acesso em 4 de dezembro de 2019.

[5] Daniel Duarte Abdala. *Sistema de coordenadas homogêneas*. 2019. Disponível em `http://www.facom.ufu.br/~abdala/GBC204/05_sistemasDeCoordenadas.pdf`. Acesso em 4 de dezembro de 2019.

[6] Abel J.P. Gomes. *Transformações Geométricas*. 2015. Disponível em `http://www.di.ubi.pt/~agomes/cg/teoricas/03-transformacoes.pdf`. Acesso em 4 de dezembro de 2019.

[7] Charles Novaes de Santana, José Garcia Vivas Miranda. *OpenGL na Computação Gráfica*. 2014. Disponível em `https://www.passeidireto.com/arquivo/24346621/tutorial-1-3-open-gl`. Acesso em 4 de dezembro de 2019.