Roteiro de Laboratório – Transformações 2D

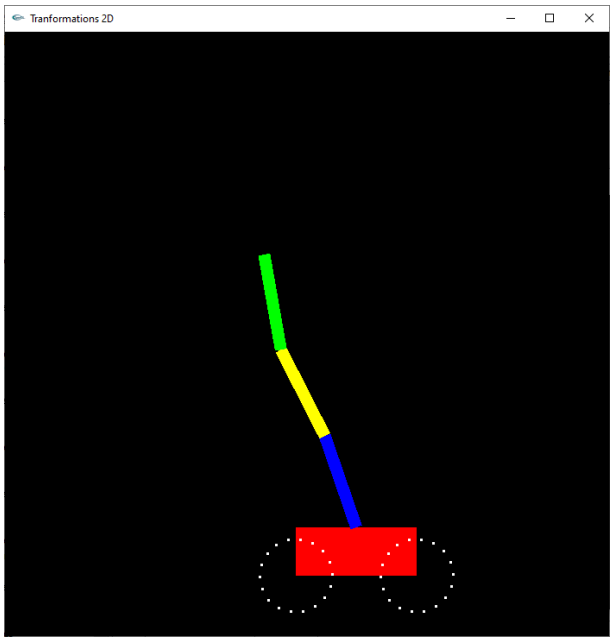
Objetivo: Trabalhar os conceitos transformações 2D, mudanças de sistema de coordenadas e animação.

Material: Sistema operacional Linux (recomendação Ubuntu 20.04) com g++, OpenGL e GLUT instalados (geralmente só são necessários dois comandos para a instalação, https://gist.github.com/AbdullahKady/f2782157991df652c2baee0bba05b788).

Antes de iniciar o roteiro, deve-se assistir a aula de OpenGL 2D!

1. Tarefa:

Criar uma aplicação usando composições de transformações para modelar os objetos. A aplicação deve criar uma janela com um robô (formado por duas rodas, um corpo e um braço articulado com 3 partes). O robô poderá se movimentar para esquerda e para direita utilizando respectivamente as teclas “a” e “d”, além de movimentar as partes dos braços em torno de suas articulações (teclas “f” e “r” movimentam a primeira haste em um sentido e no outro, as teclas “g” e “t” movimentam a segunda e as teclas “h” e “y” a terceira). Ver vídeo anexo (“Video - Aplicacao Final.mp4”) sobre o funcionamento esperado para a aplicação.



É importante salientar que a partir de agora, o conceito utilizado para desenhar os objetos mudará drasticamente em relação aos primeiros laboratórios. Com a introdução do conceito de transformações, os objetos complexos serão modelados a partir de objetos mais simples. As transformações serão utilizadas para compor os objetos e mudar sistemas de coordenadas. O robô da figura acima, por exemplo, só possui dois tipos de objetos retângulos e círculos com origens bem definidas para permitir as composições. As origens devem ser escolhidas de forma a facilitar as transformações a serem aplicadas.

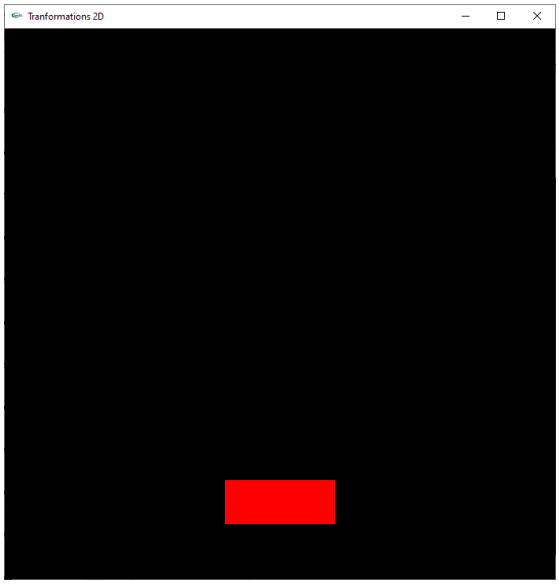
1.1. Passo: criação da janela

- Compilar os arquivos fornecidos com este roteiro com o comando “g++ -o test *.cpp -lGL -lGLU -lglut”. As bibliotecas OpenGL, GLU e GLUT são referenciadas respectivamente com os comandos -lGL -lGLU -lglut.
- Executar o arquivo compilado com o comando “./test”
- Verificar se uma janela foi criada com o fundo preto
- A implementação das chamadas dos eventos já está feita. Basta completar as funções dos arquivos cpp das classes que também já possuem os atributos (i.e., as variáveis) necessárias para representar o estado dos objetos!

1.2. Passo: criação do corpo do robô

- O primeiro passo para criar o corpo do robô é definir a função de desenho do objeto mais simples, i.e., um retângulo genérico. Para isso, é necessário escolher a origem do sistema de coordenadas desse objeto. Olhando para o robô da figura, vemos que várias transformações são feitas considerando o centro da base do retângulo, portanto, vamos escolher o centro da base como origem do nosso sistema de coordenadas do retângulo.
- Perceba que a definição dessa origem é implícita, ou seja, ela será definida a partir do momento que os pontos do retângulo formem escolhidos.
- Implemente, na classe robô, a função: *void DesenhaRect(GLint height, GLint width, GLfloat R, GLfloat G, GLfloat B);*
    - Essa função deverá criar um retângulo com a altura, largura e cores (r, g e b) passados como parâmetro e tendo o ponto (0,0) no centro de sua base.
- Implemente, na classe robô, a função: *void DesenhaRobo(GLfloat x, GLfloat y, GLfloat thetaWheel, GLfloat theta1, GLfloat theta2, GLfloat theta3);*
    - Essa função recebe a posição atual do robô (x, y), o ângulo da roda e os ângulos de cada haste, ou seja, todos os parâmetros necessários para representar o estado atual do robô.
    - Ela deverá chamar a função *DesenhaRect* passando *baseHeight* e *baseWidth* e a cor vermelha como parâmetros.
        - Se mandarmos desenhar somente com isso, o retângulo será desenhado com a base centrada no meio da janela, pois a origem da janela de visualização está no centro da janela. Teste!
    - Para colocar o retângulo da base na posição correta da tela, deve-se chamar a função *glTranslatef* antes de desenhar (antes de *DesenhaRect*) passando a posição (x, y recebidos como parâmetro e z sempre 0, dado que estamos em 2D) do robô. Isso fará uma translação da base do robô para a sua posição no mundo, ou seja, mudará o sistema de coordenada do mundo para a base do robô.
        - Se mandarmos desenhar somente com isso, o retângulo sumirá da tela, pois a função de desenho é chamada várias vezes o que faz acumular as operações de translação. Teste!
    - Para evitar o acúmulo de translações, deve-se garantir que o sistema de coordenadas foi deixado como estava ao sair desta função. Com isso, deve-se colocar um *glPushMatrix* no início (salvará o sistema de coordenadas do momento) da função e um *glPopMatrix* no final (recuperará último o sistema salvo pelo *push*).

- O resultado desta etapa deveria produzir o resultado mostrado na figura abaixo.

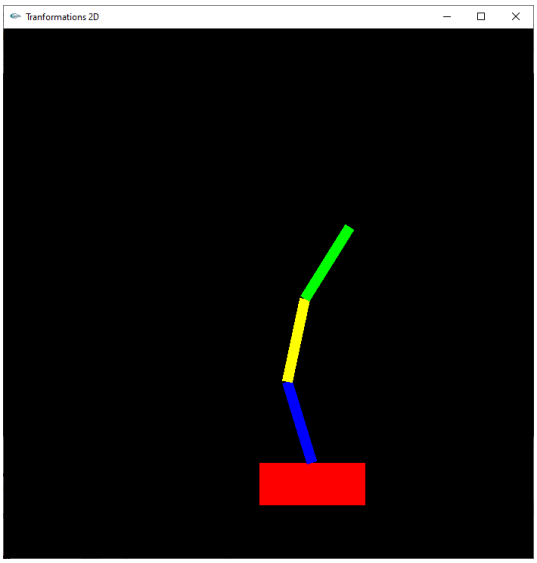


- A função *idle* chama as funções de movimentar o robô de acordo com o clique nas teclas “a” e “d”. Implemente, na classe robô, a função *MoveEmX* para movimentar o robô na horizontal.
    - Essa função só deve alterar o estado atual do robô para que ele se mova para sua nova posição de acordo com o incremento passado como parâmetro.
    - Perceba que essa função não tem nada relacionado a desenho.
    - Após essa etapa, o retângulo já deve se mover com o pressionar das teclas.

1.3. Passo: criação do braço do robô

- O braço do robô reusará a função de desenho do retângulo. Para a criação do braço, deve-se implementar a função: *void DesenhaBraco(GLfloat x, GLfloat y, GLfloat theta1, GLfloat theta2, GLfloat theta3);*
    - Essa função recebe como parâmetros a posição (x, y) do braço, assim como os três ângulos de cada haste.
    - Assumindo que não se quer sair desta função em um novo sistema de coordenadas desconhecido, é prudente colocar seu conteúdo dentro de um par *glPushMatrix* (primeiro comando da função) e *glPopMatrix* (último comando da função).
    - A primeira haste deve ser desenhada na posição dada como parâmetro (x, y), então deve-se chamar a *glTranslatef* com eles como parâmetro. Com isso, o sistema de coordenadas está na base da haste.
    - Com o sistema de coordenadas na base da haste, ela está pronta para ser rotacionada com a função *glRotatef* tendo o ângulo *theta1* para girar em torno do eixo z (0,0,1).
    - Com o sistema de condenadas rotacionado, a haste está pronta para ser desenhada.
        - Para desenhar a haste, deve-se chamar a função *DesenhaRect* passando *paddleHeight* e *paddleWidth* e a cor azul como parâmetros da primeira haste.

    - Para se desenhar a segunda haste, deve-se seguir a partir do último sistema de coordenadas usado (ou seja, o da primeira haste considerando todas as transformacoes). Com isso, basta continuar chamando as funções de transformação referentes a segunda haste, ou seja, um outro *glTranslatef* e *glRotatef*
        - O *glTranslatef* não deve deslocar nada em na horizontal (x = 0.0), porém deve subir *paddleHeight* em y antes de desenhar a próxima haste.
        - Além disso, deve-se rotacionar com o ângulo *theta2* da segunda haste.
        - Por fim, pode-se desenhar a segunda haste com a função *DesenhaRect* passando *paddleHeight* e *paddleWidth* e a cor amarela como parâmetros.
    - O mesmo procedimento pode ser feito para se desenhar a terceira haste.
- A função *DesenhaBraco* pode ser chamada dentro da função desenha robô após se desenhar o corpo.
    - Sendo chamada após o desenho do corpo, o sistema de coordenadas está no centro da base do retângulo, portanto os parâmetros de posição (x, y) devem ser relativos a essa posição (ou seja, x = 0.0 e y = baseHeight). Os ângulos podem ser apenas repassados.
    - Perceba que a função *DesenhaBraco* poderia (apesar de não muito intuitivo) ser chamada entre a função *glTranslatef* e *DesenhaRect* do corpo sem prejuízo aos resultados, dado que o sistema de coordenadas já estaria no lugar esperado e ele não está sendo alterado ao sair da *DesenhaBraco*. Teste!
    - Perceba que a função *DesenhaBraco* NÃO poderia (sem outras alterações) ser chamada antes da função *glTranslatef*, dado que o sistema de coordenadas estaria em outro lugar. Onde o braço seria desenhado? Teste!
- Neste ponto o braço já se move junto com o robô. Para fazê-lo girar basta implementar as funções *RodaBraco* 1, 2 e 3 para cada um dos ângulos. Elas serão chamadas com o pressionar das teclas mencionadas na descrição da tarefa.
    - Essas funções só devem alterar o estado atual do robô para que elas girem as hastes com o incremento passado como parâmetro.
    - Perceba que essas funções não têm nada relacionado a desenho.
    - Após essa etapa, as hastes já devem se mover com o pressionar das teclas.
- Ver resultado esperado na imagem abaixo.



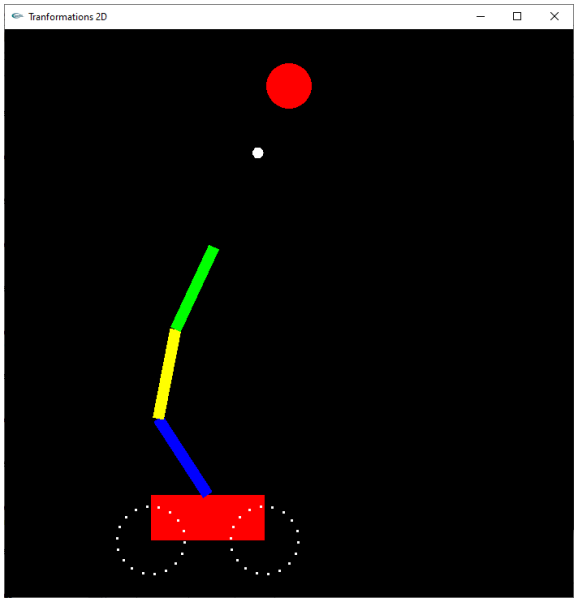
1.4. Passo: criação das rodas do robô

- O primeiro passo para criar as rodas do robô é definir a função de desenho do objeto mais simples, i.e., um círculo genérico. Para isso, é necessário escolher a origem do sistema de coordenadas desse objeto. Pensando em um círculo e suas prováveis transformações seu centro parece ser uma boa origem. Portanto, vamos escolher o centro do círculo como origem de seu sistema de coordenadas.
- Implemente, na classe robô, a função: *void DesenhaCirc(GLint radius, GLfloat R, GLfloat G, GLfloat B);*
    - Essa função deverá criar um círculo com raio *radius* e cores (r, g e b) passados como parâmetro e tendo o ponto (0,0) no seu centro.
    - Por definição, a função deverá desenhar um círculo com essas características.
    - Utilize uma sequência de pontos para representar o círculo. Pode-se desenhar um ponto a cada 20 graus até 360. A função *glPointSize* muda o tamanho do ponto para ficar mais visível.
- Implemente, na classe robô, a função: *void DesenhaRoda(GLfloat x, GLfloat y, GLfloat thetaWheel, GLfloat R, GLfloat G, GLfloat B);*
    - Essa função recebe a posição atual da roda (x, y), o ângulo da roda e a cor da roda (r, g e b), ou seja, todos os parâmetros necessários para representar o estado atual da roda.
    - Assim como no desenha braço, não se deseja alterar o sistema de coordenadas ao sair da função, portanto, é prudente colocar seu conteúdo dentro de um par *glPushMatrix* (primeiro comando da função) e *glPopMatrix* (último comando da função).
    - O primeiro passo antes de desenhar a roda é mudar o sistema de coordenadas para a posição da roda, ou seja, *glTranslatef* tendo (x, y) como parâmetros.
    - Posteriormente, a roda deve ser girada de seu ângulo, ou seja, *glRotatef* tendo *thetaWheel* como ângulo em torno de z.
    - As cores podem ser passadas adiante para a função de desenho.

- A função *DesenhaRoda* deve ser chamada duas vezes na função que desenha o robô, uma para cada roda passando para cada uma delas a posição relativa de onde a roda será desenhada no robô (ou seja, metade de largura do corpo do robô).
    - Assim como a função *DesenhaBraco*, a *DesenhaRoda* pode ser chamada em vários pontos da função desenha robô. Teste o que funciona e veja se faz sentido!
- Neste ponto, o robô se move, mas as rodas estão fixas (veja resultado esperado na figura da tarefa). Para fazê-las girar, deve-se atualizar o ângulo da roda *gThetaWheel*.
    - Isso poderia ser feito com um incremento qualquer ao mover o robô (em *MoveEmX*). Porém, muito provavelmente isso causaria um efeito *Moon Walker* ou *Papaleguas*. Teste!
    - Para evitar isso, pode-se calcular o incremento angular baseado na distância dada como incremento em X, ou seja, fazendo a regra de três para achar o incremento angular para *gThetaWheel* em função do incremento *dx* usado para gX.
- Agora o robô já deve se mover como esperado!

2. Tarefa:

Incrementar a tarefa anterior para fazer o braço atirar projeteis circulares (círculo branco na imagem) a partir de sua haste mais externa (haste verde na imagem) ao apertar a tecla de espaço. Pode travar o tiro de um novo projetil até que o anterior não seja mais útil, ou seja, não precisa permitir atirar várias vezes seguidas. O projetil poderá atingir o alvo e nesse caso, o alvo deverá mudar de cor (alternando entre vermelho, verde e azul) e de posição (aleatória na janela, mas sempre na mesma altura). Ver vídeo anexo ("Video - Aplicacao Final.mp4") sobre o funcionamento esperado para a aplicação.



2.1. Passo: criação da funcionalidade de tiro

- A funcionalidade de tiro é controlada em dois lugares
  - Sua criação (se já não tem um ativo) é feita ao apertar a tecla espaço
  - Seu movimento é atualizado na *idle* de acordo com a sua trajetória
- Para se desenhar o tiro, pode-se utilizar uma função similar a que desenhou a roda, porém utilizando um polígono fechado ao invés de pontos.
- A função *DesenhaTiro* recebe a posição do tiro deve simplesmente desenhá-lo (utilizando a *DesenhaCirc* apropriada com o raio do tiro *radiusTiro*) após trocar seu sistema de coordenadas para a posição (x, y) do tiro.
  - Pensar se é necessário utilizar o envoltório push e pop.
- Assumindo que quando um tiro é criado ele tem uma posição atual e uma direção, pode-se atualizar o seu estado continuamente e independente de quem o criou.
  - Para isso, basta implementar a função: *void Move();*
    - Ela deverá atualizar a posição atual do tiro com base na sua posição anterior (*gX*, *gY*), sua direção (*gDirectionAng*) e sua velocidade de incremento (*gVel*).
- Para o tiro não ficar se movendo para sempre, mesmo sem ter mais utilidade (ex., já estando longe da janela de visualização), pode-se destruí-lo. A função *idle* destrói o tiro que não é mais útil (ou não é mais válido) para que outro possa ser criado.
  - Implemente a função: *bool Valido();*

        - Ela deverá retornar falso para tiros que já tiverem percorrido uma distância maior do um máximo definido *DISTANCIA_MAX*.
- A classe robô é responsável por criar o tiro com a função de atirar
    - Para isso, implemente, na classe robô, a função: *Tiro* Atira();*
        - Ele deverá criar um tiro utilizando o estado atual do robô para calcular a posição e direção do tiro em relação a ponta e direção da última haste.
        - Essa função deverá calcular a posição da ponta e da base da última haste, para que se tenha acesso a posição (ponta da última haste) e direção do tiro (ângulo do vetor indo da base até a ponta da haste).
            - A ponta da haste é representada por x = 0.0 e y = *paddleHeight* após aplicar todas as transformações usadas para desenhá-la.
            - A base da haste é representada por x = 0.0 e y = *0.0* após aplicar todas as transformações usadas para desenhá-la.
            - As transformações usadas foram: rotação de *gTheta3*, translação de *paddleHeight* em y, rotação de *gTheta2*, translação de *paddleHeight* em y, rotação de *gTheta1*, translação de *baseHeight* em y e translação para a posição no mundo *gX* e *gY*.
            - DICA1: usar *atan2* para o ângulo e não *atan*
            - DICA2: criar uma função auxiliar para fazer a rotação de um ponto (x, y), ou seja, replicar o comportamento da matriz de rotação.
- Seu robô agora já pode atirar. Verifique se funciona como esperado.

2.2. Passo: criação da funcionalidade de alvo

- A funcionalidade de alvo é controlada basicamente na *idle* para verificar se ele foi atingido. Quando o alvo é atingido, ele é recriado em uma nova posição e com uma cor diferente.
- Implemente a função: *void Recria(GLfloat x, GLfloat y);*
    - Ela deve atualizar a posição do alvo e incrementar a cor *gColor* para o próximo valor possível entre (0, 1 e 2);
- Implemente a função: *bool Atingido(Tiro *tiro);*
    - Ela deve retornar verdadeiro se o tiro estiver dentro do raio do alvo *radiusAlvo*;
- O desenho do alvo pode ser feito de maneira similar à do tiro (utilizando desta vez o *radiusAlvo*), porém a cor deve ser condicionada ao valor de *gColor* (ou seja, deve receber vermelho, verde ou azul dependendo respectivamente dos valores 0, 1 ou 2 de *gColor*).
- Agora sua aplicação está completa.
    - Teste também a funcionalidade de animação do robô ao apertar a tecla “1”.
    - Verifique como foi implementada.

2.3. Passo: correção do tempo entre chamadas *idle*

- Apesar de completa, sua aplicação assume que o tempo decorrido entre duas chamadas *idle* é constante e idêntico entre máquinas. Porém, ele não é e pode ficar estragar os cálculos de velocidade se as máquinas forem diferentes (ou seja, vai ficar mais rápido ou devagar).
    - Teste a sua aplicação da seguinte forma: atire e enquanto o tiro estiver se movendo, arraste a janela para outra posição. Enquanto estiver arrastando, a *idle* não será

chamada e, com isso, não terá atualização do movimento do tiro. Portanto, o tiro continuará de onde parou.

- Teste também a simulação de uma máquina mais lenta: coloque um comando *for* vazio (ex. *for(int i = 0; i < 90000000; i++);*) dentro da *idle*. Isso irá reduzir o número de chamadas da *idle* como se fosse uma máquina mais lenta. Portanto, o tiro ficará bem mais lento do que da forma original. Contudo, se fosse uma aplicação para o usuário final, ela deveria rodar com o movimento do tiro seguindo uma unidade de tempo independente da velocidade da máquina.

- Para corrigir esse problema, utilize a função *glutGet(GLUT_ELAPSED_TIME);* para obter o tempo decorrido do início do programa.
  - Coloque o código seguinte no início da *idle* e utilize a diferença de tempo *timeDiference* para corrigir o movimento do tiro, ou seja, passe *timeDiference* para a função move do tiro e multiplique pelo cálculo do incremento da distância (distância = velocidade * tempo).

```
static GLdouble previousTime = glutGet(GLUT_ELAPSED_TIME);
GLdouble currentTime, timeDiference;
//Pega o tempo que passou do inicio da aplicacao
currentTime = glutGet(GLUT_ELAPSED_TIME);
// Calcula o tempo decorrido desde de a ultima frame.
timeDiference = currentTime - previousTime;
//Atualiza o tempo do ultimo frame ocorrido
previousTime = currentTime;
```

  - Agora, com ou sem o comando *for* vazio o tiro se moverá com a mesma velocidade. Porém, com o comando *for*, a atualização do movimento será mais espaçada.

2.4. Passo: colocando um placar de contagem na aplicação

- Para colocar um placar contando o número de vezes que o alvo foi atingido na sua aplicação, utilize o código abaixo.

```
int atingido = 0;
static char str[1000];
void * font = GLUT_BITMAP_9_BY_15;
void ImprimePlacar(GLfloat x, GLfloat y)
{
   glColor3f(1.0, 1.0, 1.0);

   //Cria a string a ser impressa
   char *tmpStr;
   sprintf(str, "Atingido: %d", atingido );
   //Define a posicao onde vai comecar a imprimir
   glRasterPos2f(x, y);
   //Imprime um caractere por vez
   tmpStr = str;
   while( *tmpStr ){
      glutBitmapCharacter(font, *tmpStr);
      tmpStr++;
   }
}
```

- A variável *atingido* representa o número de vezes que o alvo foi atingido e deve ser incrementada no local apropriado na *idle*.

- A função *ImprimePlacar* deve ser chamada dentro da *renderScene* para imprimir o placar na posição (relativo à janela de visualização) desejada.
    - Teste os valores da posição de impressão.
- Veja o resultado esperado na figura abaixo.