PEF – 5743 – Computação Gráfica Aplicada à Engenharia de Estruturas

Prof. Dr. Rodrigo Provasi

e-mail: provasi@usp.br

Sala 09 – LEM – Prédio de Engenharia Civil

Bibliotecas Gráficas: Fundamentação Teórica

- Até aqui vimos as bibliotecas e diversas ferramentas que auxiliam na criação e geração de objetos tridimensionais.
- Esses objetos são representados e utilizados das mais diversas formas em um código de computação gráfica.
- Essa seção veremos toda a fundamentação teórica para a representação dos sólidos, geração de malhas, iluminação e comunicação necessários para que haja um bom funcionamento do programa.

Eventos e entradas de usuário

- Antes de adentrarmos na parte mais matemática das representação de sólidos, vamos lidar com os eventos de entrada de usuário.
- Como em programa que traz computação gráfica, em geral, os objetos não são estáticos, ele permite interações com o usuário, respondendo adequadamente aos comandos.

Eventos e entradas de usuário - Delegate

- Para explicar melhor como funciona os eventos, um conceito de linguagem orientada à objetos: *delegate*.
- Da mesma forma que existem ponteiros para variáveis, é possível criar um ponteiro para uma função. Esse ponteiro é chamado de *delegate.*
- Assim, pode-se deixar um ponteiro para uma função e apontar a função de acordo com a programação feita.
- Por exemplo, é possível definir em uma classe um *delegate* chamado OnKeyPress e quando for trabalhar com essa classe definir para qual função esse *delegate* apontará.

Eventos e entradas de usuário - Delegate

- No C# existe o conceito de *event* que no fundo funciona como uma lista de *delegates*, permitindo assim que não apenas uma, mas várias funções em sequência. Dessa forma, pode-se definir um conjunto de ações a serem executadas.
- Com esses conceitos é possível associar à classe que lida com os eventos e o desenho na tela com o que se deseja realizar.

Eventos e entradas de usuário - Delegate

Orientation

Reading the mouse position is easy :

```
// Get mouse position
int xpos, ypos;
glfwGetMousePos(&xpos, &ypos);
```

but we have to take care to put the cursor back to the center of the screen, or it will soon go outside the window and you won't be able to move anymore.

```
// Reset mouse position for next frame
glfwSetMousePos(1024/2, 768/2);
```

Notice that this code assumes that the window is 1024*768, which of course is not necessarily the case. You can use glfwGetWindowSize if you want, too.

We can now compute our viewing angles :

```
// Compute new orientation
horizontalAngle += mouseSpeed * deltaTime * float(1024/2 - xpos );
verticalAngle   += mouseSpeed * deltaTime * float( 768/2 - ypos );
```

Eventos e entradas de usuário - Delegate

Position

The code is pretty straightforward. By the way, I used the up/down/right/left keys instead of the awsd because on my azerty keyboard, awsd is actually zqsd. And it's also different with qwerZ keyboards, let alone korean keyboards. I don't even know what layout korean people have, but I guess it's also different.

```
// Move forward
if (glfwGetKey( GLFW_KEY_UP ) == GLFW_PRESS){
    position += direction * deltaTime * speed;
}
// Move backward
if (glfwGetKey( GLFW_KEY_DOWN ) == GLFW_PRESS){
    position -= direction * deltaTime * speed;
}
// Strafe right
if (glfwGetKey( GLFW_KEY_RIGHT ) == GLFW_PRESS){
    position += right * deltaTime * speed;
}
// Strafe left
if (glfwGetKey( GLFW_KEY_LEFT ) == GLFW_PRESS){
    position -= right * deltaTime * speed;
}
```

Eventos e entradas de usuário - Delegate

```
do{

    // Clear the screen
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Use our shader
    glUseProgram(programID);

    // Compute the MVP matrix from keyboard and mouse input
    computeMatricesFromInputs();
    glm::mat4 ProjectionMatrix = getProjectionMatrix();
    glm::mat4 ViewMatrix = getViewMatrix();
    glm::mat4 ModelMatrix = glm::mat4(1.0);
    glm::mat4 MVP = ProjectionMatrix * ViewMatrix * ModelMatrix;

    // Send our transformation to the currently bound shader,
    // in the "MVP" uniform
    glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);

    // Bind our texture in Texture Unit 0
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, Texture);
    // Set our "myTextureSampler" sampler to use Texture Unit 0
    glUniform1i(TextureID, 0);
```

```
    // 1rst attribute buffer : vertices
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
    glVertexAttribPointer(
        0,                  // attribute. No particular reason for 0, but must match the layout in the shader.
        3,                  // size
        GL_FLOAT,           // type
        GL_FALSE,           // normalized?
        0,                  // stride
        (void*)0            // array buffer offset
    );

    // 2nd attribute buffer : UVs
    glEnableVertexAttribArray(1);
    glBindBuffer(GL_ARRAY_BUFFER, uvbuffer);
    glVertexAttribPointer(
        1,                                // attribute. No particular reason for 1, but must match the layout in the shader.
        2,                                // size : U+V => 2
        GL_FLOAT,                         // type
        GL_FALSE,                         // normalized?
        0,                                // stride
        (void*)0                          // array buffer offset
    );

    // Draw the triangle !
    glDrawArrays(GL_TRIANGLES, 0, 12*3); // 12*3 indices starting at 0 -> 12 triangles

    glDisableVertexAttribArray(0);
    glDisableVertexAttribArray(1);

    // Swap buffers
    glfwSwapBuffers(window);
    glfwPollEvents();

} // Check if the ESC key was pressed or the window was closed
while( glfwGetKey(window, GLFW_KEY_ESCAPE ) != GLFW_PRESS &&
       glfwWindowShouldClose(window) == 0 );
```

Eventos e entradas de usuário - Override

- Outra maneira de fazer com que o código execute o que se pretende é fazendo *overrides* da classe original e definindo o comportamento de acordo com o que se deseja. Na próxima figura, as funções Start e Update são sobrescritas para executar as funções desejadas. Isso permite que o controle seja feito pela classe derivada.

Eventos e entradas de usuário - Override

```
public class EnemyAI : MonoBehavior
{
  // These values will appear in the editor, full properties will not.
  public float Speed = 50;
  private Transform _playerTransform;
  private Transform _ myTransform;
  // Called on startup of the GameObject it's assigned to.
  void Start()
  {
    // Find some gameobject that has the text tag "Player" assigned to it.
    // This is startup code, shouldn't query the player object every
    // frame. Store a ref to it.
    var player = GameObject.FindGameObjectWithTag("Player");
    if (!player)
    {
      Debug.LogError(
        "Could not find the main player. Ensure it has the player tag set.");
    }
    else
    {
      // Grab a reference to its transform for use later (saves on managed
      // code to native code calls).
      _playerTransform = player.transform;
    }
    // Grab a reference to our transform for use later.
    _myTransform = this.transform;
  }
```

```
  // Called every frame. The frame rate varies every second.
  void Update()
  {
    // I am setting how fast I should move toward the "player"
    // per second. In Unity, one unit is a meter.
    // Time.deltaTime gives the amount of time since the last frame.
    // If you're running 60 FPS (frames per second) this is 1/60 = 0.0167,
    // so w/Speed=2 and frame rate of 60 FPS (frame rate always varies
     // per second), I have a movement amount of 2*0.0167 = .033 units
    // per frame. This is 2 units.
    var moveAmount = Speed * Time.deltaTime;
    // Update the position, move toward the player's position by moveAmount.
    _myTransform.position = Vector3.MoveTowards(_myTransform.position,
      _playerTransform.position, moveAmount);
  }
}
```

Transformações

- Em computação gráfica, muitas vezes é necessária a utilização de transformações, seja para aplicação de efeitos de animação, para facilitar criação de objetos ou até mesmo para o simples posicionamento dos mesmos em uma cena.
- Os principais tipos de transformações existentes são: translação (*translate*), rotação (*rotate*), escala (*scale*) e cisalhamento (*shear*).

Translação

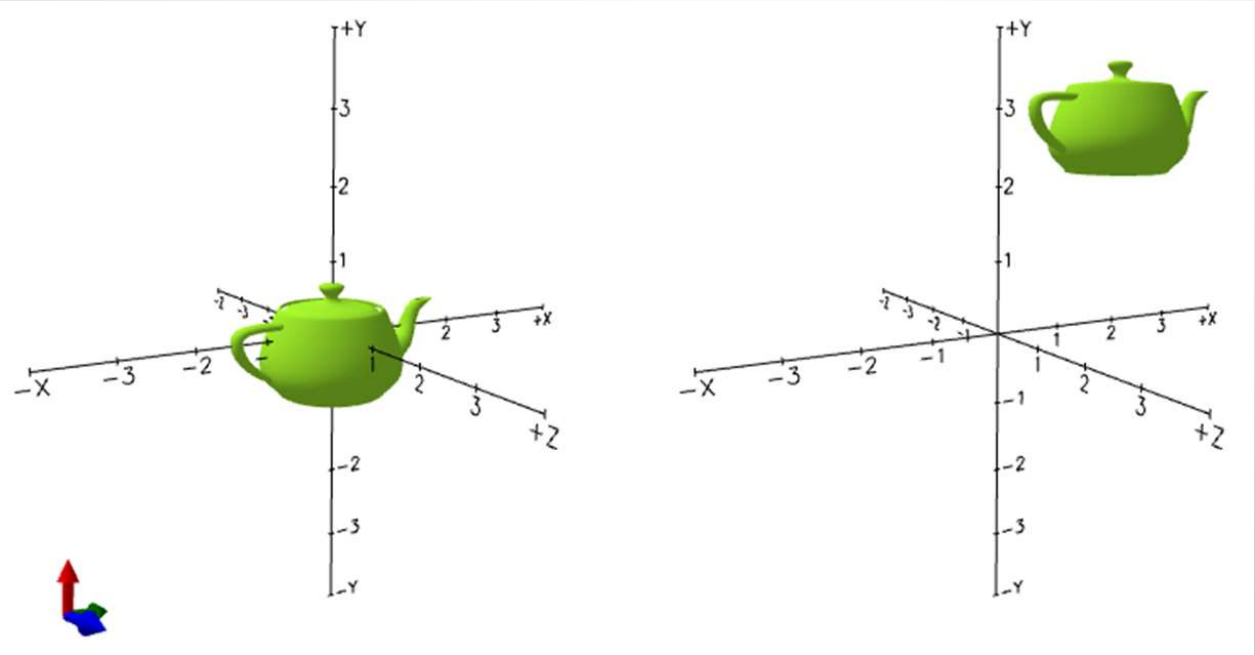
- A transformação de translação consiste em mover um ponto (ou objeto) no espaço. Para tal, deve-se apenas acrescentar um deslocamento em suas coordenadas.

$$P = [x \quad y \quad z]^T; T = [\Delta x \quad \Delta y \quad \Delta z]^T; P' = [x' \quad y' \quad z']^T$$

$$P' = P + T$$

$$[x' \quad y' \quad z']^T = [x + \Delta x \quad y + \Delta y \quad z + \Delta z]^T$$

Translação

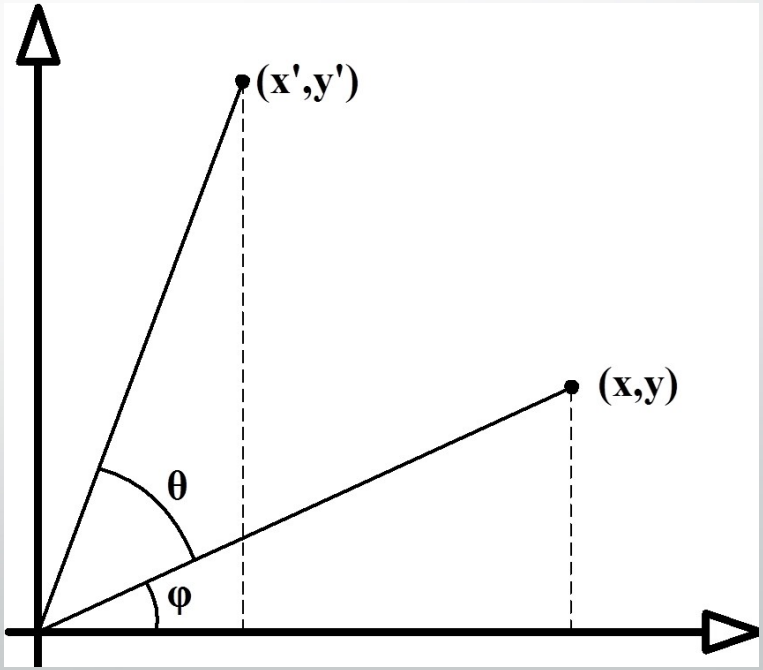


Rotação

- A transformação de rotação consiste em rotacionar um vetor (definido pelo ponto final, já que se assume que esse está ligado a origem) ou um objeto em torno de um centro de rotação e um eixo de rotação por um determinado ângulo.

Rotação

- Dado um ponto $P = [x \quad y \quad z]^T$ como extremo de um vetor, aplicando-se uma rotação em torno do eixo Z com centro de rotação na origem e um ângulo θ:



Rotação

$$x' = r\cos\varphi\cos\theta - r\,\text{sen}\,\varphi\,\text{sen}\,\theta$$
$$y' = r\cos\varphi\,\text{sen}\,\theta + r\,\text{sen}\,\varphi\cos\theta$$
$$z' = z$$

Como:

$$x = r\cos\varphi$$
$$y = r\,\text{sen}\,\varphi$$

Tem-se:

$$x' = x\cos\theta - y\,\text{sen}\,\theta$$
$$y' = x\,\text{sen}\,\theta + y\cos\theta$$

Rotação

- Utilizando-se da representação matricial:

$$P = [x \quad y \quad z]^T; \boldsymbol{R_z} = \begin{bmatrix} \cos\theta & -\text{sen}\,\theta & 0 \\ \text{sen}\,\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}; P' = [x' \quad y' \quad z']^T; P' = \boldsymbol{R_z} P$$

Rotação

- Similarmente, as rotações em torno do eixo X e Y possuem as seguintes matrizes de rotação, respectivamente:

$$\boldsymbol{R_x} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \text{sen}\,\theta \\ 0 & -\text{sen}\,\theta & \cos\theta \end{bmatrix}$$

$$\boldsymbol{R_y} = \begin{bmatrix} \cos\theta & 0 & -\text{sen}\,\theta \\ 0 & 1 & 0 \\ \text{sen}\,\theta & 0 & \cos\theta \end{bmatrix}$$

Rotação

- Para o caso em que se deseja fazer uma rotação com um centro de rotação diferente da origem, deve-se primeiramente aplicar uma transformação de translação, de modo que o centro de rotação fique na origem, aplicar a rotação, e aplicar novamente uma translação de modo a retornar à posição anterior.

Rotação

- Dado um centro de rotação $C = [c_x \quad c_y \quad c_z]^T$, uma rotação em torno do eixo Z resultaria em:

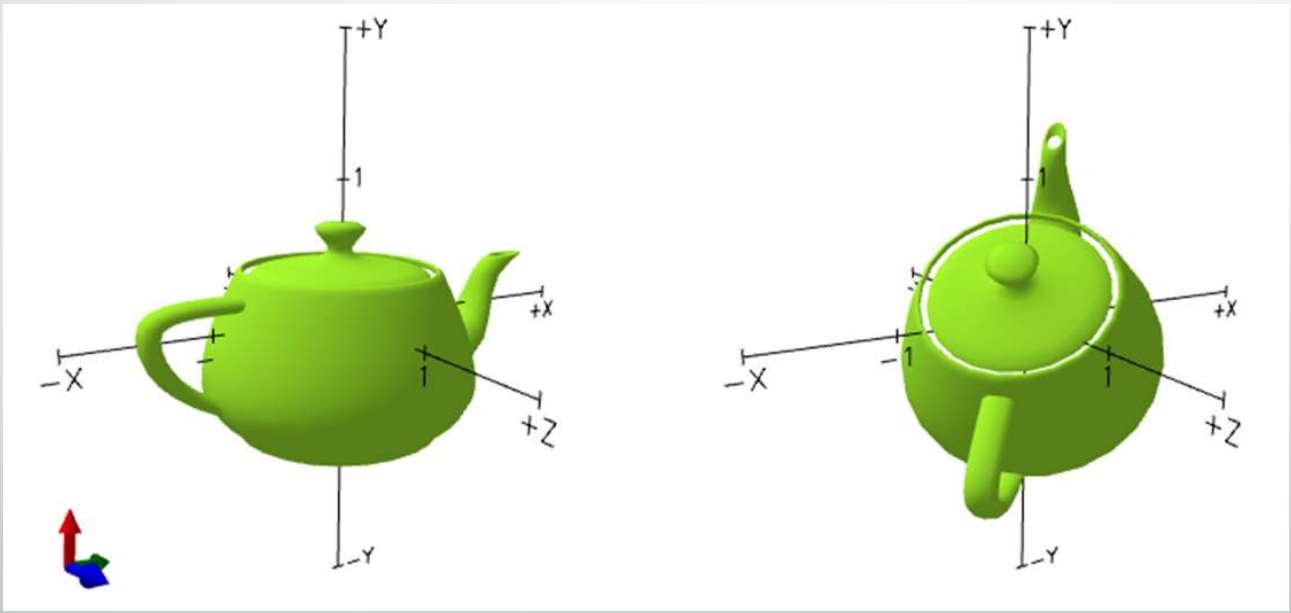
$$P = [x \quad y \quad z]^T; P' = [x' \quad y' \quad z']^T; \boldsymbol{R_z} = \begin{bmatrix} \cos\theta & -\text{sen}\,\theta & 0 \\ \text{sen}\,\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix};$$

$$T = [-c_x \quad -c_y \quad -c_z]^T; T' = [c_x \quad c_y \quad c_z]^T; P' = \boldsymbol{R_z}(P + T) + T'$$

Rotação

- Caso seja desejada uma rotação em torno de um eixo genérico, esta operação pode ser decomposta em três rotações, em torno dos eixos conhecidos (X, Y e Z), tomando-se a devida atenção para a ordem das operações, uma vez que a transformação de rotação não é comutativa.

Rotação



Escala

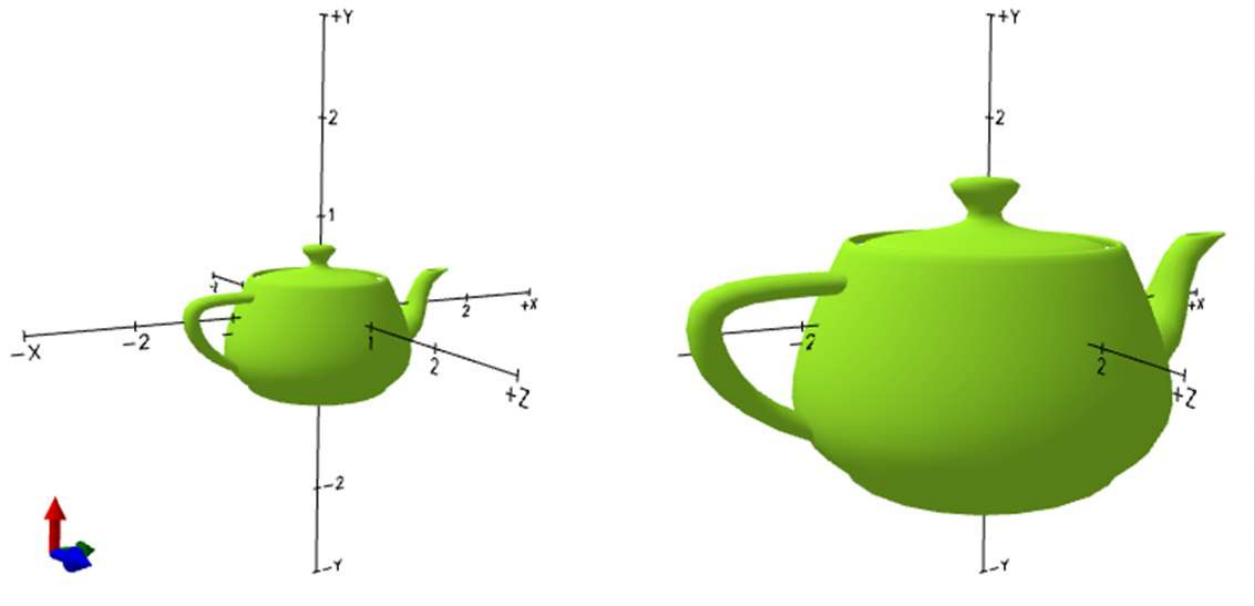
- Dado um ponto P e os fatores de escala s_x,s_y e s_z, têm-se:

$$x' = s_x x \; ; \; y' = s_y y \; ; \; z' = s_z z$$

- E em sua representação matricial:

$$P = [x \quad y \quad z]^T ; \boldsymbol{S} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} ; P' = [x' \quad y' \quad z']^T ; P' = \boldsymbol{S}P$$

Escala



Cisalhamento

- A transformação de cisalhamento consiste em distorcer um objeto, como se houvesse um deslizamento entre suas camadas. Aplicando-se uma transformação de cisalhamento com um fator s_h na coordenada x, proporcional à coordenada y a um ponto P, têm-se:

$$x' = x + s_h y$$

$$y' = y$$

$$z' = z$$

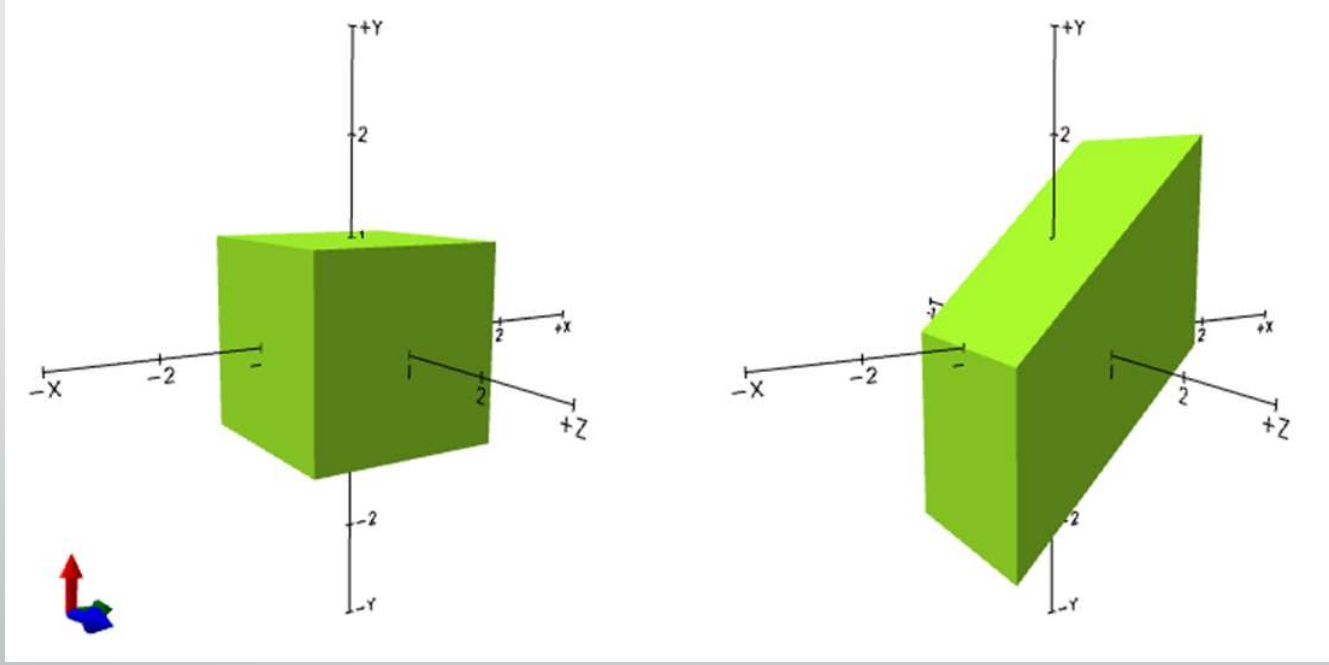
Cisalhamento

- Esta transformação também pode ser representada em forma de uma matriz:

$$\boldsymbol{S} = \begin{bmatrix} 1 & s_h^{xy} & s_h^{xz} \\ s_h^{yx} & 1 & s_h^{yz} \\ s_h^{zx} & s_h^{zy} & 1 \end{bmatrix}$$

onde s_h^{mn} é o fator de cisalhamento da coordenada m proporcional à coordenada n.

Cisalhamento



Coordenadas Homogêneas

- Pode-se observar que as transformações de rotação, escala e cisalhamento podem ser aplicadas a partir das matrizes de transformação apresentadas, porém, a translação é aplicada somando-se o vetor das distâncias.
- Com o objetivo de se realizar todas as transformações com multiplicações e agrupar diversas transformações em uma única matriz, foi criado o conceito de coordenadas homogêneas, que consiste em acrescentar uma dimensão ao problema em questão. Desta forma, um ponto P no espaço tridimensional, em coordenadas homogêneas ficaria:

$$P = [x \quad y \quad z \quad 1]^T$$

Coordenadas Homogêneas

- Com isto, para se realizar uma translação $\boldsymbol{T}$, basta fazer a multiplicação de um ponto em coordenadas homogêneas pela matriz:

$$\boldsymbol{T} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Coordenadas Homogêneas

- As demais transformações vistas anteriormente utilizando-se coordenadas homogêneas são dadas por:

$$\boldsymbol{R_x} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\text{sen}\,\theta & 0 \\ 0 & \text{sen}\,\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \boldsymbol{R_y} = \begin{bmatrix} \cos\theta & 0 & \text{sen}\,\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\text{sen}\,\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Coordenadas Homogêneas

$$\boldsymbol{R_z} = \begin{bmatrix} \cos\theta & -\text{sen}\,\theta & 0 & 0 \\ \text{sen}\,\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \boldsymbol{S} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \boldsymbol{T}$$

$$= \begin{bmatrix} 1 & s_h^{xy} & s_h^{xz} & 0 \\ s_h^{yx} & 1 & s_h^{yz} & 0 \\ s_h^{zx} & s_h^{zy} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Composição de Transformações

- Tendo-se todas as transformações sendo realizadas por multiplicações, é possível a realização de diversas transformações com uma única matriz.
- Isto é muito vantajoso, pois uma vez que se tem uma grande quantidade de pontos, para os quais é necessária a mesma sequência de transformações, pode ser calculada a matriz de transformação equivalente a todas as transformações e fazer apenas uma multiplicação para cada ponto, reduzindo-se consideravelmente a quantidade de processamento necessário para o cálculo da posição final de todos os pontos.

Composição de Transformações

- Por exemplo, se desejamos aplicar uma transformação de escala S e em seguida uma de translação T ao ponto P, podemos calcular a matriz de transformação equivalente M da seguinte maneira:

$$P' = \boldsymbol{TS}P \rightarrow P' = \boldsymbol{M}P$$

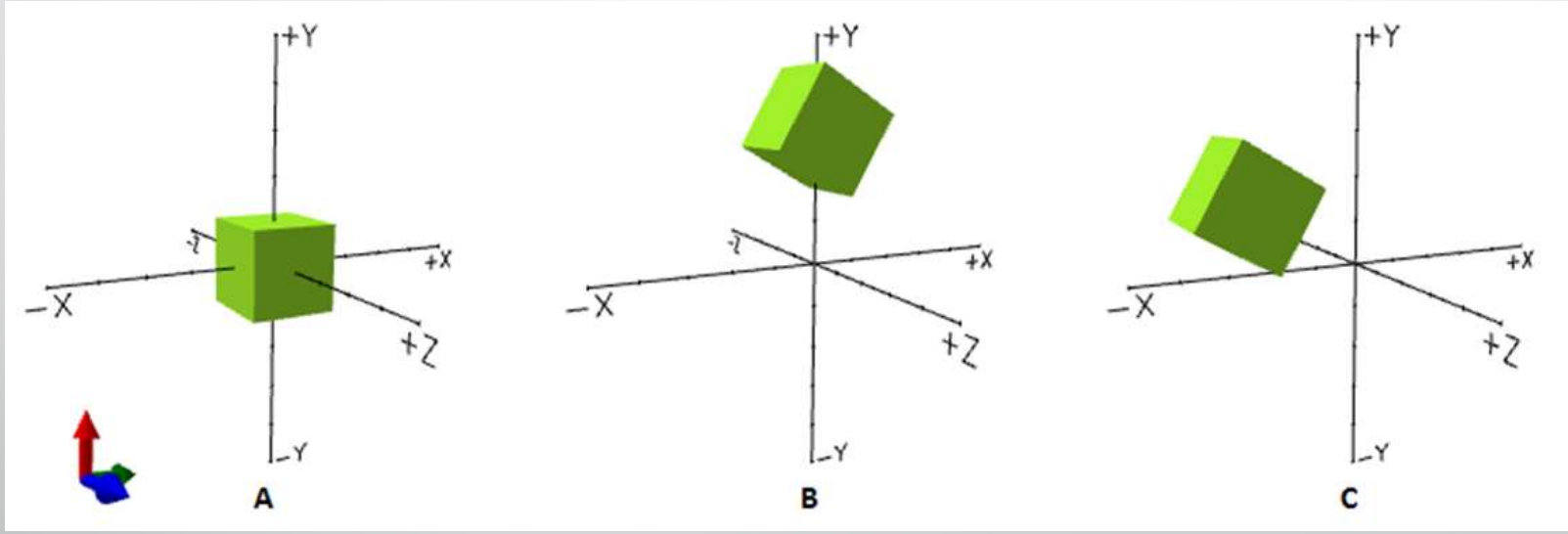
$$\boldsymbol{M} = \begin{bmatrix} s_x & 0 & 0 & t_x \\ 0 & s_y & 0 & t_y \\ 0 & 0 & s_z & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Composição de Transformações

- Quando se deseja aplicar diversas operações de transformação em um ponto, é importante observar que, embora a multiplicação de matrizes seja associativa, a mesma não é comutativa, ou seja, a ordem em que as operações são feitas é importante para o resultado final, uma vez que ao alterarmos a ordem, teremos resultados diferentes.
- No caso anterior, se fosse aplicada a transformação de translação antes da de escala, a matriz resultante desta operação seria:

$$\boldsymbol{M} = \begin{bmatrix} s_x & 0 & 0 & s_x t_x \\ 0 & s_y & 0 & s_y t_y \\ 0 & 0 & s_z & s_z t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Composição de Transformações



A imagem A corresponde a posição original, a imagem B corresponde a aplicação de uma rotação seguida de uma translação, enquanto a imagem C representa a translação aplicada antes da rotação.