Lab4. Perspective projection

Introduction

A 3D projection (or graphical projection) is a design technique used to display a three-dimensional (3D) object on a two-dimensional (2D) surface. These projections rely on visual perspective and aspect analysis to project a complex object for viewing capability on a simpler plane. This has the effect that **distant objects appear smaller than nearer objects**.

Mathematically speaking, perspective projection or perspective transformation is a linear projection where three dimensional objects are projected on a picture plane.

OpenGL consists of two general classes of projection transformations: orthographic (parallel) and perspective.

Orthographic Projections

Orthographic, or parallel, projections consist of those that involve **no perspective correction**. There is **no adjustment for distance** from the camera made in these projections, meaning objects on the screen will appear the **same size** no matter how close or far away they are.

To setup this type of projection we use the OpenGL provided `glOrtho()` function.

```
glOrtho(left, right, bottom, top, near, far);
```

Where `left` and `right` specify the x-coordinate **clipping planes**, `bottom` and `top` specify the y-coordinate clipping planes, and `near` and `far` specify the distance to the z-coordinate clipping planes. Together these coordinates provide a box shaped viewing volume.

Perspective Projections

Perspective projections create **more realistic looking scenes**, so that's what you will most likely be using most often.

In perspective projections, as an object gets farther from the viewer it will appear smaller on the screen- an effect often referred to as foreshortening.

The viewing volume for a perspective projection is a **frustum**, which looks like a pyramid with the top cut off, with the narrow end toward the user.

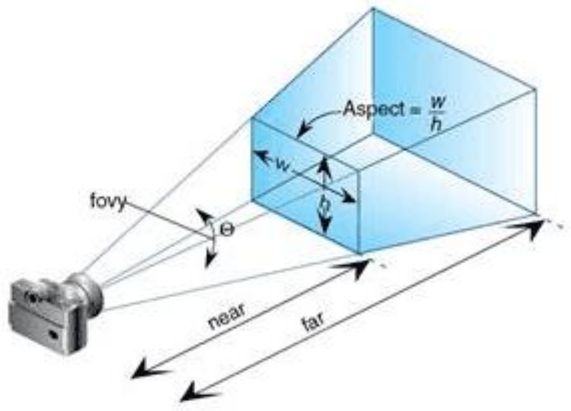
To setup the view frustum, and thus the perspective projection, we have two possible choices: `glFrustum` and `gluPerspective`.

```
glFrustum(left, right, bottom, top, near, far);
```

Using `glFrustum` enables us to specify an **asymmetrical** frustum, which can be very useful in some instances, but isn't what we typically want to do.

```
gluPerspective(fov, aspect, near, far);
```

The `gluPerspective` function specifies a **symmetrical** viewing frustum into the world coordinate system. `fov` specifies, in degrees, the angle ***in the y direction*** that is visible to the user (**field of view**); `aspect` is the **aspect ratio** of the scene, which is width divided by the height. This will determine the field of view ***in the x direction***. `near` and `far` specify the distance to the z-coordinate clipping planes (***always positive***).

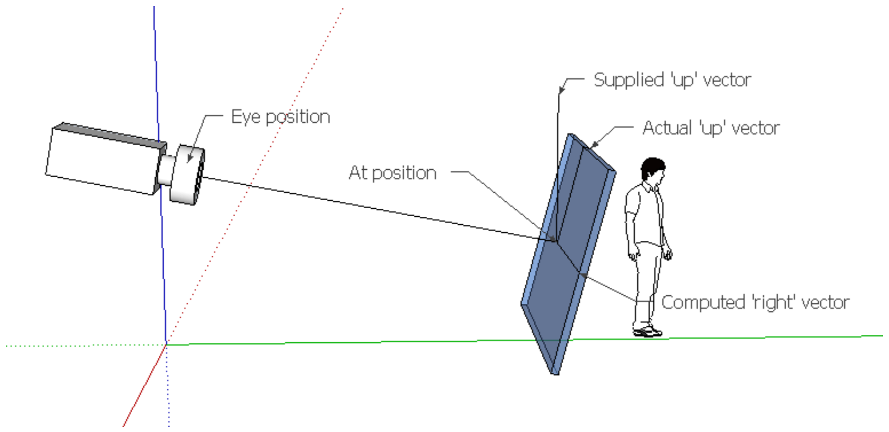


In order to view the perspective projection from the beginning of the drawing process, it is preferable to put the call to the function `gluPerspective` in the **reshape callback** function (triggered when a window is reshaped or immediately before a window's first display callback after a window is created or whenever an overlay for the window is established). Recall that the reshape callback must be set/registered in the main function using the `glutReshapeFunc` function.

Before calling the `gluPerspective` function we must set the current matrix mode to GL_PROJECTION using the `glMatrixMode` function. After `glMatrixMode`, we must call the function `glLoadIdentity()` in order to set the matrix at the top of the stack in the matrix stack as the identity matrix, so that any previous transformation will not affect the subsequent changes.

The position of the virtual camera i.e. the "HOW" and from "WHERE" to view the scene are configured using the function `gluLookAt` that sets an **eye point**, a **reference point** indicating the center of the scene, and an **up vector**.

```
gluLookAt(eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ)
```



The default values of the arguments of the function `gluLookAt` are (0, 0, 0, 0, 0, -1, 0, 1, 0). These values may not be suitable for a given scenario so a call to the function `gluLookAt` must be made with the appropriate arguments.

Finally, we must specify the area of the window where the drawing region should be put into. This is done calling the `glViewport` function that fixes the bottom left corner of the drawing region (coordinates argument (x, y)), and the number of pixels the drawing region should go in each direction (dimensions arguments (width, height)).

```
glViewport(x, y, width, height)
```

Mathematically speaking, `glViewport` specifies the affine transformation of x and y from normalized device coordinates to window coordinates. Let x_{nd} and y_{nd} be the normalized device coordinates. Then the window coordinates x_w and y_w are computed as follows:

$$x_w = x_{nd} + 1\,\frac{width}{2} + x$$

$$y_w = y_{nd} + 1\,\frac{height}{2} + y$$

For example, `glViewport(0, 0, width, height)` call tells OpenGL to map the lower left corner of the projected image to the coordinate (0,0) of the OpenGL window; It also specifies that the upper right corner of the projected image is mapped to the coordinate (width, height) in window space; this is the upper right corner of the OpenGL window.

A complete example of drawing a 3D wireframe cone with perspective projection is the following:

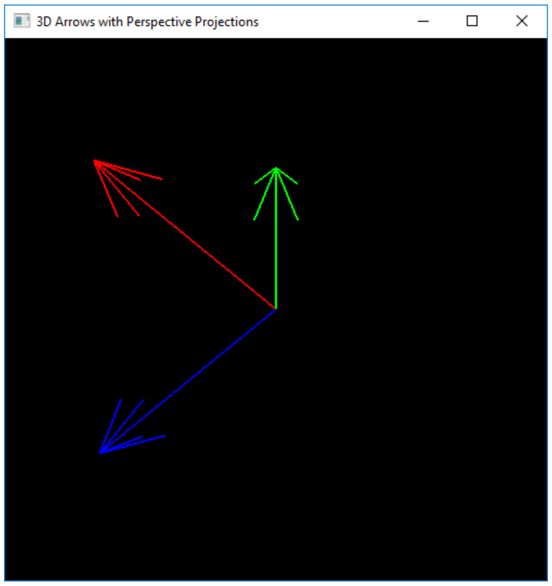
```
def render_Scene():
    # Draw a wireframe red cone
    glColor3f(1.0,0.0,0.0);
    glutWireCone(0.8,1,50,50)

def reshape(x,y):
    # Set a new projection matrix
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(20.0,x/y,4.5,20.0);
    gluLookAt(2,-4,-2,0,0,0,0,1,0);
    glViewport(0,0,x,y);  # Use the whole window for rendering
```

Exercices

- Test the previous example on your machine and tune the parameters of the functions `gluPerspective` and `gluLookAt` and note their influence of the final view of the scene.
- Test the previous code with a solid version of the rendered object (cone). What is missing to have a realistic view of the scene?

- Draw three arrows (red, green and blue) in the directions of x, y and z –axis respectively (the arrows start at the origin (0,0,0) and end at (length,0,0), (0,length,0) and (0,0,length) respectively where `length` is a parameter to define at the beginning of the code (as global variable).
- For each arrow add two arrowheads (each composed of two small segments), each in the planes in which the arrow is located. For example, the x-axis arrow should have its arrowheads in the XY and XZ planes. The dimension of the arrow should be set through a parameter `arrow_dim` set at the beginning of the code.
- Make a perspective projection of the scene (the arrows) and tune the parameters of the functions `gluPerspective` and `gluLookAt` so that you can see a clear view of the whole scene with an acceptable size of the three arrows (like the one in the following figure).



Homework

- Add some animation to the previous code by performing 3D rotations about the 3 coordinate axes (x, y and z) and add the subsequent modifications that allow to have a clear view of the whole scene with an acceptable size of the three arrows.
- Modify the previous code to replace the arrows shape with a shape of your choice.
- What will happen if we combine the rotation process with a translation without any control on it? Propose a solution.