Outline

1. The View Frustum
2. Creating the Projection Matrix
3. Orthogonal Projections
4. Positioning the Camera
5. Creating the View Matrix
6. Controlling the Camera
7. Assignment
The View Frustum

Definition (Frustum)
A **frustum** is a truncated pyramid.

Definition (The View Frustum)
The **view frustum** is the region of world coordinate space that contains the part of the scene that will be rendered.

- The view frustum is bounded by six planes.
  - Left and right planes
  - Top and bottom planes
  - Near and far planes
Creating the View Frustum

- The function `gluPerspective()` establishes the size and shape (but not the position) of the view frustum.
- It takes four parameters.
  - The vertical view angle.
  - The aspect ratio (width/height).
  - The distance to the near plane.
  - The distance to the far plane.
Creating the View Frustum

Side view of the view frustum.
Creating the View Frustum

Front view of the view frustum, from the eye point.
Creating the View Frustum

- The aspect ratio is the width divided by the height.
- Typical aspect ratios are $4/3$ and $5/4$.
- For example, if the screen has a resolution of $1024 \times 768$, then its aspect ratio is $4/3$.

```cpp
gluPerspective(45.0, 4.0/3.0, 1.0, 1000.0);
```
Creating the View Frustum

- The view frustum for a perspective projection may also be created using the function `glFrustum()`.

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

- `left`, `right`, `top`, and `bottom` are the $x$ and $y$ boundary values at the near plane.
- `near` and `far` are always given as positive distances from the viewpoint.
Orthogonal Projections

- The view frustum produces a perspective view on the screen.
  - The eye is at the center of the projection.
- On the other hand, an orthogonal projection projects along parallel lines.
  - It is as though the view point is at infinity.
Orthogonal Projections

- To create an orthogonal projection, use `gluOrtho()` instead of `gluPerspective()`.

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

- Again, `near` and `far` are always given as positive distances from the viewpoint.
- `left`, `right`, `top`, and `bottom` are the `x` and `y` coordinates of the planes.
Example (Perspective and Orthogonal Projections)

- The code.
- The executable.
The function `gluLookAt()` positions the view frustum in space.

It takes nine parameters, representing two points and a vector, expressed in world coordinates.

- The eye point, or position of the camera.
- The look point.
- The up vector, or orientation.
In eye coordinates,
- The eye point is at $(0, 0, 0)$,
- The look point is $(0, 0, -1)$,
- The up vector is $(0, 1, 0)$.

The `gluLookAt()` function computes the transformation matrix from world coordinates to eye coordinates.
Positioning the View Frustum

- In world coordinates,
  - The eye point is wherever we want the camera to be.
  - The look point is often the origin.
  - The up vector is almost always $(0, 1, 0)$.

```c
gluLookAt(5.0, 2.0, 5.0,
          0.0, 0.0, 0.0,
          0.0, 1.0, 0.0);
```
Positioning the View Frustum

- In world coordinates,
  - The **eye point** is wherever we want the camera to be.
  - The look point is often the origin.
  - The up vector is almost always \((0, 1, 0)\).

```c
gluLookAt(5.0, 2.0, 5.0,
          0.0, 0.0, 0.0,
          0.0, 1.0, 0.0);
```
In world coordinates,
- The eye point is wherever we want the camera to be.
- The look point is often the origin.
- The up vector is almost always \((0, 1, 0)\).

```c
gluLookAt(5.0, 2.0, 5.0,
          0.0, 0.0, 0.0,
          0.0, 1.0, 0.0);
```
Positioning the View Frustum

- In world coordinates,
  - The eye point is wherever we want the camera to be.
  - The look point is often the origin.
  - The up vector is almost always $(0, 1, 0)$.

```c
gluLookAt(5.0, 2.0, 5.0,
          0.0, 0.0, 0.0,
          0.0, 1.0, 0.0);
```
The View Matrix

- The `gluLookAt()` function creates the **view matrix** and multiplies the current matrix by it.
- The result is the **modelview matrix**.
- In a literal sense, it “moves” the entire scene, thereby creating the illusion that the camera has moved.
For this reason, it is important to call `gluLookAt()` 
- after loading the identity matrix and 
- before performing any other transformations.

Typically, this is one of the first things done in the `display()` function.
Controlling the Camera Position

- The camera may be movable or fixed.
- If it is movable, then it is usually controlled by spherical coordinates with the look point at the center.
  - Distance from the look point \( \text{camDist} \).
  - Pitch angle \( \text{camPitch} \).
  - Yaw angle \( \text{camYaw} \).
Controlling the Camera Position

The following formulas compute the $x$, $y$, and $z$ coordinates of the camera.

\[
\begin{align*}
  x &= r \cos \varphi \sin \theta \\
  y &= r \sin \varphi \\
  z &= r \cos \varphi \cos \theta
\end{align*}
\]

where $r = \text{distance}$, $\varphi = \text{pitch angle}$, and $\theta = \text{yaw angle}$.
Controlling the Eye Position

Example (Controlling the Eye Position)

// Convert degrees to radians
float yaw = camYaw*PI/180.0;
float pitch = camPitch*PI/180.0;

// Compute rectangular coordinates
float eye.x = camDist*cos(pitch)*sin(yaw);
float eye.y = camDist*sin(pitch);
float eye.z = camDist*cos(pitch)*cos(yaw);

// Position the camera
gluLookAt(eye.x, eye.y, eye.z,
        look.x, look.y, look.z,
        0.0, 1.0, 0.0);
The keyboard() Function

Example (The keyboard() Function)

```c
void keyboard(unsigned char key, int x, int y)
{
    switch (key)
    {
        case '+': case '=':
            camDist /= zoomFactor;
            break;
        case '-': case '_':
            camDist *= zoomFactor;
            break;
    ...
}
glutPostRedisplay();
return;
}
```
void special(int key, int x, int y)
{
    switch (key)
    {
    case GLUT_KEY_LEFT:
        camYaw -= yawIncr;
        break;
    case GLUT_KEY_RIGHT:
        camYaw += yawIncr;
        break;
    
    glutPostRedisplay();
    return;
    }
Controlling the Camera Position

Example (Controlling the Camera Position)

- The code.
- The executable.
Controlling the Look Point

- In a similar way we can control the look point instead of the camera location.
- The mouse to make the camera to pan left, right, up, or down.
- The + and – keys move the camera (and the look point) forward or backward.
- How do we calculate the \( x \), \( y \), and \( z \) coordinates of the look point?
Controlling the Look Point

Example (Controlling the Look Point)

- The code.
- The executable.
Homework

- Read Section 2.6 – orthographic viewing.
- Read Sections 5.1 - 5.2 – perspective viewing.