Basics for Enhanced Visualization: 3D/Data

3D Rendering

Rodrigo Cabral

Polytech Nice - Data Science

cabral@unice.fr

Source: by Gilles Tran - `http://www.oyonale.com/modeles.php?lang=en&page=40`, Public Domain, `https://commons.wikimedia.org/w/index.php?curid=745313`.

Outline

These slides are partly inspired from Lionel Fillatre's slides.

Summary

- Object surfaces are tessellated into simple primitives.
- Draw primitives with glBegin()/glEnd() blocks.
  - glVertexf() , glColorf() , glNormalf().
- Nested transformation blocks:
  - glPushMatrix() , glPopMatrix() , glLoadMatrix() , glMultMatrix().
- **This class:** rasterization, view port, image formation in OpenGL(Model View mode) and the Z-buffer.

Rasterization

How to project on the image buffer?

- Problem: given a triangle, how do you color the pixels that it covers?

Rasterization

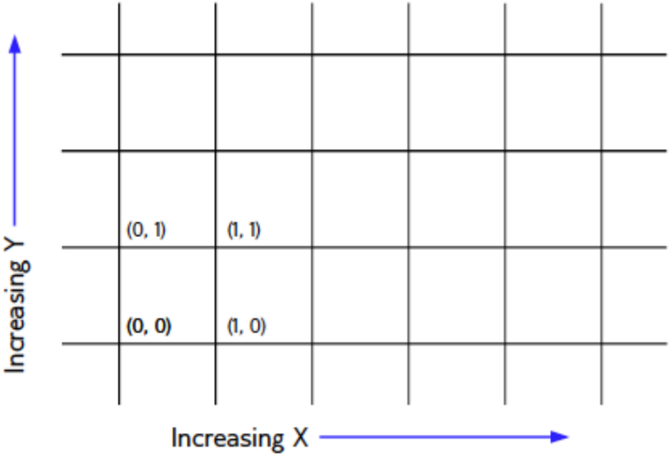
How to project on the image buffer?

- Problem: given a triangle, how do you color the pixels that it covers?

  Solution: given in two steps
  - Project the triangle to screen space.
  - Compute which pixels are covered by the projection.

Rasterization

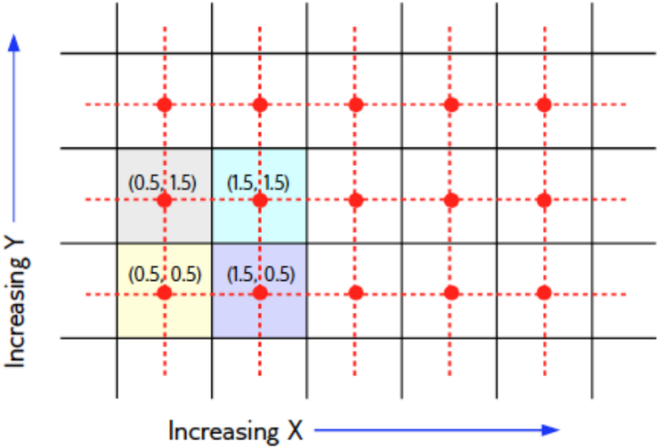
Pixel coordinates in OpenGL framebuffer



Rasterization

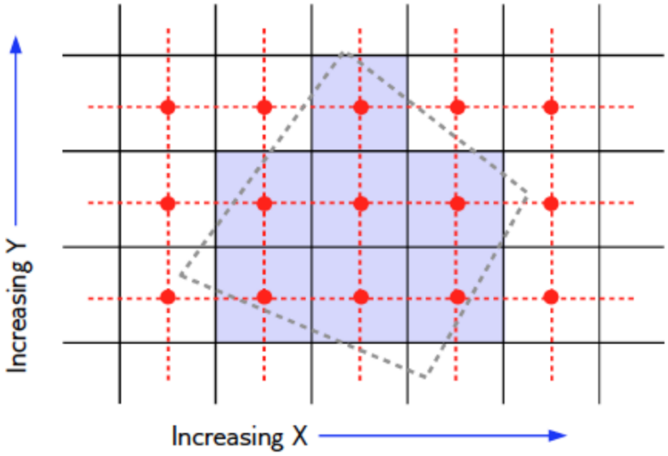
Pixel coordinates in OpenGL framebuffer

- Pixel centers are at half-integer coordinates.



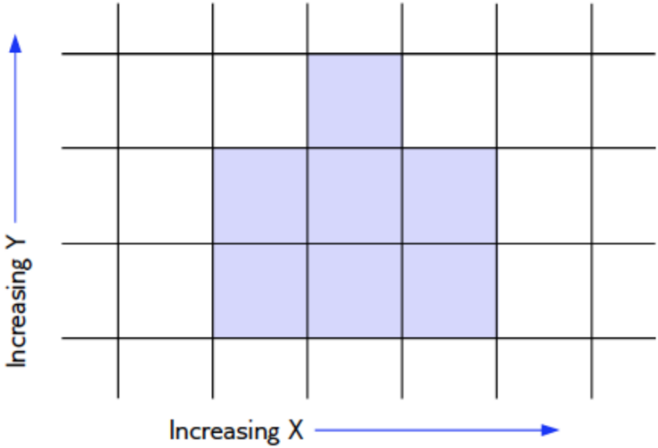
Rasterization rules: area primitives

- Output a **fragment** (a pixel from a primitive) if pixel center is inside area.



Rasterization rules: area primitives

- **Combine** fragment color with existing pixel color.



What does "combine" mean?

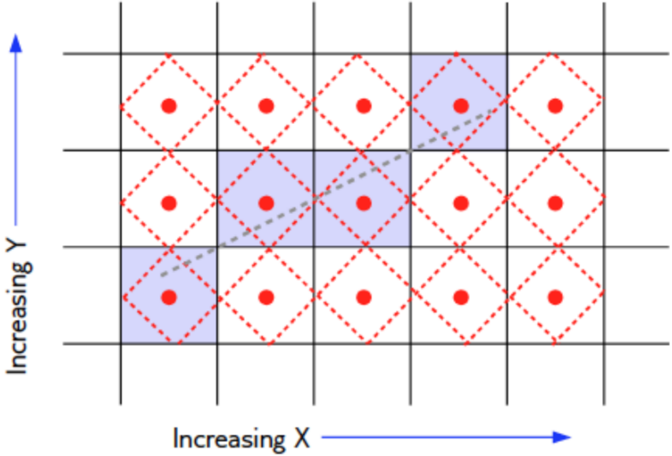
- Tipically we compare the fragment depth against the z-buffer (we will talk about it later) and replace the existing pixel if the fragment is closer.
- For other specific effects, we can
  - use other tests.

What does "combine" mean?

- Tipically we compare the fragment depth against the z-buffer (we will talk about it later) and replace the existing pixel if the fragment is closer.
- For other specific effects, we can
  - use other tests.
  - **Blend** the fragment color with an existing pixel color instead of replacing it.
  - **Blending** combined with back-to-front rendering allows transparency effects.
  - This is the purpose of the α channel in **glColor4f**.
  - To enable blending for transparency we use the commands **glEnable(BLEND)** and **glBlendFunc(`GL_SRC_ALPHA,GL_ONE`)**.

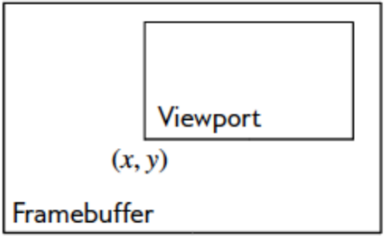
Rasterization rules: line primitives

- Output a fragment if line intercepts "diamond".



Specifying the viewport

- The active section of frame buffer is called **viewport**.
- **glViewport(x,y,w,h)**: (x, y) are the coordinates in pixels on the screen of the origin of the framebuffer (lower-left point) and (w, h) are its width and height in pixels.
- The viewport is initially set to the entire screen.



Viewport

Normalized device coordinates

- The viewport is always mapped to the interval $[-1, 1]^2$.
- Projection is handled in a consistent way independent of framebuffer characteristics.
- OpenGL handles the mapping from normalized coordinates to pixel coordinates.

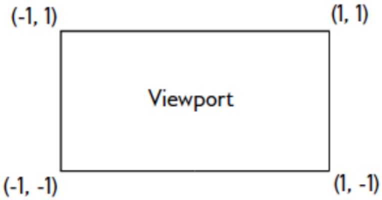
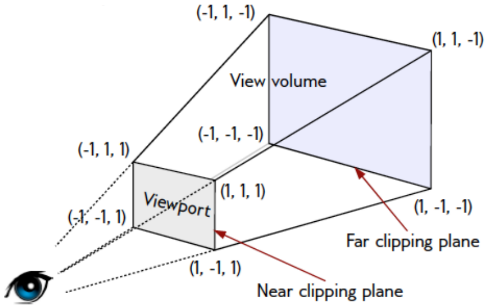


Image formation in OpenGL

View volume or frustum

- Visible part of the scene is the frustum of a pyramid.



- All objects are mapped to normalized device coordinates in the interval $[-1, 1]^3$.
- Everything outside is discarded.

Image formation in OpenGL

Projective transformation

- Maps view volume to $[-1, 1]^3$.
- Viewer is assumed to be looking along $-z$ axis.

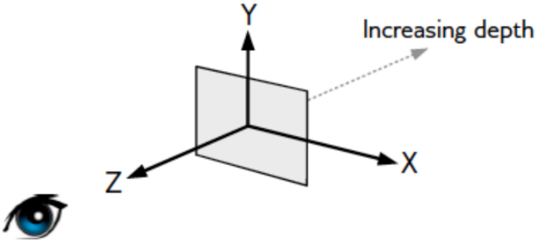
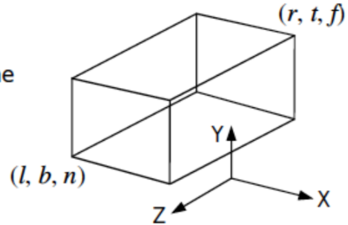


Image formation in OpenGL

Orthographic projection

- Object appears to be the same size regardless of distance.
- View volume is assumed to be the following

$$
\begin{aligned}
x = l &\equiv \text{left plane} \\
x = r &\equiv \text{right plane} \\
y = b &\equiv \text{bottom plane} \\
y = t &\equiv \text{top plane} \\
z = n &\equiv \text{near plane} \\
z = f &\equiv \text{far plane}
\end{aligned}
$$



- It does not correspond to our vision, but it is extensively used in computer assisted design systems $\Longrightarrow$ sizes are independent of depth!

Image formation in OpenGL

Orthographic projection

- Projection matrix in homogeneous coordinates is given by

$$\mathbf{P}_o = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Orthographic projection

- Projection matrix in homogeneous coordinates is given by

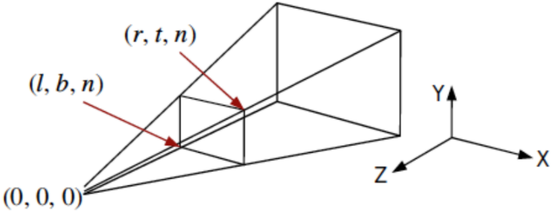
$$\mathbf{P}_o = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- After transforming to Cartesian coordinates, the point is in the normalized interval $[-1, 1]^3$.
- The top two elements in Cartesian coordinates are the normalized pixel coordinates and the third component keeps track of depth information.

Image formation in OpenGL

Perspective projection

- Rays converge at eye assumed to be at origin.
- Points (l, b, n) and (r, t, n) define the near clipping plane and the viewing volume is defined as follows



- This is the usual perspective transformation from image formation.

Image formation in OpenGL

Perspective projection

- Projection matrix in homogeneous coordinates is given by

$$\mathbf{P}_p = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

- Note that due to normalization requirements this matrix is different from the perspective projection in 3D vision.

Camera transformation

- The last missing piece is to align the camera with the viewing direction.
- Camera reference frame is specified (in world coordinates) by

- the eye position **e**,
- the gaze direction **g**,
- the view up vector **t**,
- neither **g** nor **t** need to be unit vectors and they do not need to be orthogonal.

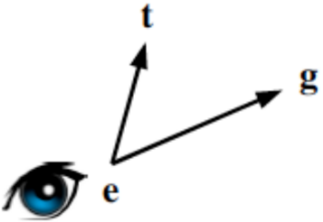
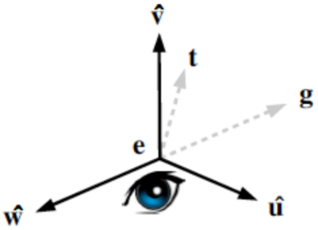


Image formation in OpenGL

Camera transformation

- We construct an orthonormal basis $[\hat{\mathbf{u}}\ \hat{\mathbf{v}}\ \hat{\mathbf{w}}]$ from $\mathbf{g}$ and $\mathbf{t}$:

- $\hat{\mathbf{w}} = \dfrac{\mathbf{g}}{\|\mathbf{g}\|}$,
- $\hat{\mathbf{u}} = \dfrac{\mathbf{t} \times \hat{\mathbf{w}}}{\|\mathbf{t} \times \hat{\mathbf{w}}\|}$,
- $\hat{\mathbf{v}} = \hat{\mathbf{u}} \times \hat{\mathbf{w}}$,
- These vectors plus translation **e** form the camera reference frame.



Camera transformation

- The camera transformation (same as $\mathbf{K}_{wc}$ in standard image formation) is given by a translation composed with a rotation:

$$\mathbf{K}_{wc} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ w_x & w_y & w_z & 0 \\ v_x & v_y & v_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Image formation in OpenGL

Camera transformation

- The camera transformation (same as $\mathbf{K}_{wc}$ in standard image formation) is given by a translation composed with a rotation:

$$\mathbf{K}_{wc} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ w_x & w_y & w_z & 0 \\ v_x & v_y & v_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

1. Set the 3D model position and orientation in the world reference frame with **glMatrixMult**($\mathbf{K}_{ow}$) where $\mathbf{K}_{ow}$ is the corresponding transformation matrix.
2. Transform everything to the camera reference frame **glMatrixMult**($\mathbf{K}_{wc}$).

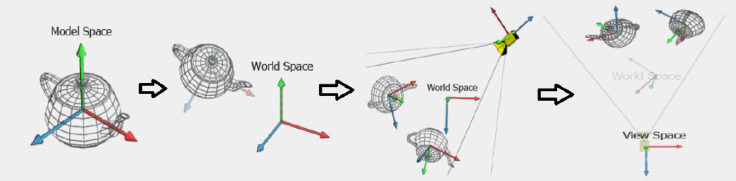
2' **glMatrixMult**($\mathbf{K}_{wc}$) can be done with the command **gluLookAt(e, c, t)** where **c** is the center of the observed scene.

Image formation in OpenGL

Model transformation

- The equivalent of the extrinsic matrix for 3D scene modelling in OpenGL is the model-view matrix:

$$\mathbf{M}_v = \mathbf{K}_{wc}\mathbf{K}_{ow}$$



Full image formation pipeline

- Every object is transformed by:

$$\mathbf{M} = \mathbf{P}_p\mathbf{M}_v \text{ (or } \mathbf{M} = \mathbf{P}_o\mathbf{M}_v\text{)}$$

Full image formation pipeline

- Every object is transformed by:

$$\mathbf{M} = \mathbf{P}_p \mathbf{M}_v \text{ (or } \mathbf{M} = \mathbf{P}_o \mathbf{M}_v\text{)}$$

- OpenGL maintains multiple current transformation matrices and stacks:
  - one for the model view matrix $\mathbf{M}_v$
  - and one for the projection matrix $\mathbf{P}$

Full image formation pipeline

- Every object is transformed by:

$$\mathbf{M} = \mathbf{P}_p\mathbf{M}_v \text{ (or } \mathbf{M} = \mathbf{P}_o\mathbf{M}_v\text{)}$$

- OpenGL maintains multiple current transformation matrices and stacks:
  - one for the model view matrix $\mathbf{M}_v$
  - and one for the projection matrix $\mathbf{P}$
- To modify, push or pop $\mathbf{M}_v$, use the command **glMatrixMode(`GL_MODELVIEW`)**.
- To modify, push or pop $\mathbf{P}$, use the command **glMatrixMode(`PROJECTION`)**.

Occluded objects and the Z-buffer

Hidden surface removal

- Near objects hide or **occlude** objects that are behind them.

Source: Elliot Erwitt, New York, 2000.

Occluded objects and the Z-buffer

Hidden surface removal

- Color at a rendered pixel depends primarily on the nearest object at that point.
- Naive solution: sort and render objects back to font - painter's algorithm.
    - Inefficient.
    - Not as easy as it sounds, see the picture!

Source: André Kertész, *Arm and Ventilator* New York, 1937.

Other solutions

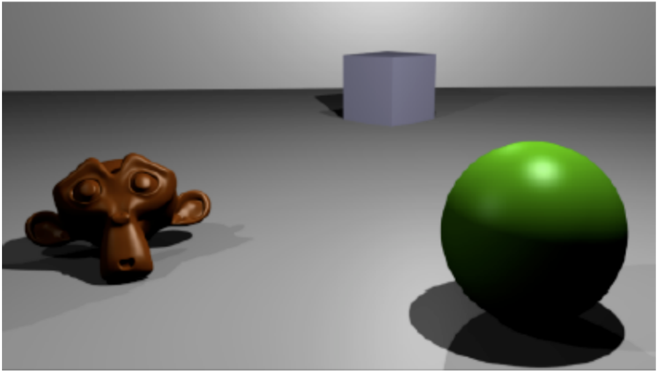
- Raytracing: trace a ray through the pixel to see which object is hit first.
- Z-buffer:
  - Draw objects one by one in any order.
  - At each pixel store closest depth value and corresponding fragment seen so far.
    - As the depth is in the camera z axis direction, we call it **Z-buffer** method.
  - At pixel **p**, let an object have color **c** and depth d (given by the negative of the third component of the projection):
    - if $d < d_{\text{old}}$ at **p**:

$$d_{\text{old}} := d \text{ and } c_{\text{old}} := c$$

Occluded objects and the Z-buffer

Z-buffer example

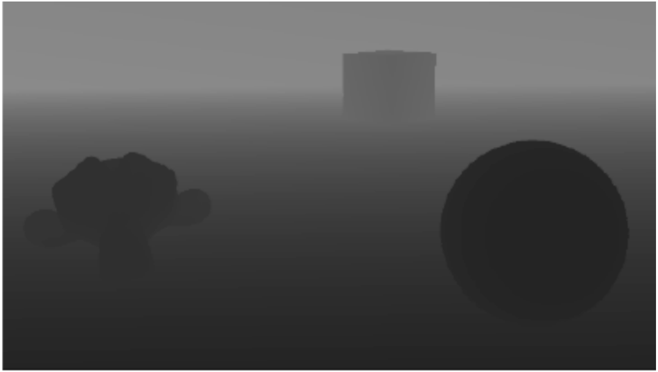
- Rendered 3D scene



Occluded objects and the Z-buffer

Z-buffer example

- Corresponding final Z-buffer values (dark: near, light: far)



Raytracing or Z-buffer

- Raytracing:
  1. Requires specialized hardware for acceleration.
  2. Requires special libraries for coding: not supported directly neither in OpenGL, neither in Direct3D (windows only API).
  3. It can simulate very realistic lighting patterns using laws of optics.

  $\Longrightarrow$ **Used mainly in offline applications.**

Occluded objects and the Z-buffer

Raytracing or Z-buffer

- Raytracing:
  1. Requires specialized hardware for acceleration.
  2. Requires special libraries for coding: not supported directly neither in OpenGL, neither in Direct3D (windows only API).
  3. It can simulate very realistic lighting patterns using laws of optics.

  $\Longrightarrow$ **Used mainly in offline applications.**

- Z-buffer:
  1. Easy hardware acceleration.
  2. Functions already exist both in OpenGL and Direct3D.
  3. It handles only simple lighting models (Phong model). Difficult to implement shadows, refractions, reflections.

  $\Longrightarrow$ **Used in most real-time applications** (games, augmented/virtual reality).

Conclusions

- 3D rendering is a Big Data problem when considering real-time constraints.
- It is efficiently solved using GPU and programming in OpenGL.
- OpenGL describes an object in primitives then projects the primitives on the frame buffer in a similar process as in image formation.
- The projections at each pixel are called fragments. Thus each primitive may have multiple fragments.
- Fragment processing: decide what is the color for each pixel.
  - It depends on depth of objects: Z-buffer.
  - It depends on the texture of the primitive **(next class)**.
  - It depends on the lighting sources **(next class)**.