Review of Tuesday

- Projection matrix transforms viewing volume into canonical volume

- Matrix can be created by finding linear relationships between coordinate axes
Review of Tuesday

• Clipping is performed before \textit{perspective division}.

• Clipping intersects polygon edges with planes of viewing volume.

• Efficient clipping performs simple tests before computing actual intersections (bounding boxes, convex hulls, outcodes, etc.).
A Note on Planes and Normals

- Planes can be fully specified by three points
- These points may be used to specify a triangle
- Use the cross product to find ‘third orthogonal vector’
- Normalized vector corresponds to triangle/plane normal

\[ n \cdot (p - p_0) = 0 \]
Clipping and Primitive Assembly

- Primitives are assembled from vertices and connectivity information (OpenGL: GL_LINES, GL_TRIANGLES,…)

- After clipping, geometry is contained in [-w,w]^3
- Perspective division produces canonical volume [-1,1]^3

- Work of rasterizer is now limited to what is within camera frustum
Mapping Results to the Screen

- Projection Matrix only specifies mapping to 2D (clip-coordinates)
- After clipping, we have normalized device coordinates (x,y coordinates lie in [-1,1]^2)
- We need to specify **viewport** parameters (target region on screen where contents of 2D image plane are displayed)

```c
glViewport(x,y,width,height) //target region (in pixels) on window
```

OpenGL
Viewport Transform

- From **normalized device coordinates** to **window/screen coordinates**
Different Viewports

- Viewport properties specify mapping to window/screen coordinates, which are relevant during rasterization.
Pipeline Summary

Object Coordinates → Vertex Processor → Clipper and Assembler → Rasterizer → Fragment Processor

Clip Coordinates → Clipped Triangles

Normalized Device Coordinates → 2D Triangles

Window Coordinates → Fragments
Adding Depth

- How we perceive depth
  - Parallax (displacement of objects during camera movement)
  - Stereoscopy (3D from two 2D images)
  - Perspective (angles and size)
  - Color attenuation (fog, medium)
  - Depth of Field (DOF)
Adding Depth

- How do we render images that contain objects at different depths?

- Objects in the back have to appear behind objects in the front

- Solve the hidden-surface removal problem
Painter’s algorithm

• An object-space approach to the hidden-surface removal problem
  • Sort objects based on their depth values (depth sort)
  • Render objects back to front, painting over objects that were drawn previously

desired scene
Painter’s algorithm

- Problems of depth sort
  - Objects/Primitives have to be sorted every time camera changes
  - Objects may have overlapping depth ranges
Painter’s algorithm

• Problems of depth sort
  • Objects may be intersecting
Adding Depth

- Fortunately, there is some data that we can use:
  - We know depth of vertices from its normalized device coordinates (contains a depth (z-value) in [-1,1])
  - The rasterizer interpolates depth values (just as color, texture coordinates, etc.) of vertices to fragments

- Every fragment has a depth value
  - If we store depth values of fragments that are drawn into the frame buffer, we can do a comparison to discard other incoming fragments – this is called **z-buffering** or **depth-buffering**
The Z-Buffer

- **Z-buffer algorithm** is the most widely used hidden-surface removal technique
- The standard frame buffer (color values) is supplemented by a depth-buffer (z-buffer)
- Stores depth value of the most recently drawn fragment
- No longer performs object-space depth sort

- Default:
  - Incoming fragments are rejected, if depth value is larger than value present in depth-buffer (because it would be occluded)
  - Otherwise: Update color and frame buffer
The Z-Buffer

• Several fragments created by rasterizing primitives can map to the same target pixel
• Compare and update value in the z-buffer (resolve depth-ordering problem per fragment)
The Z-Buffer

- The z-buffer has a resolution and depth
- Resolution typically corresponds to image-plane resolution
- Depth (bits per pixel) defines how well we can resolve depth differences

bad depth precision can lead to z-fighting
Back-face Culling

• We can use other tricks in 3D to avoid that multiple fragments ‘fight’ for the same pixel
Back-face Culling

- **Back-face culling** discards triangles that face away from the camera (represent the back of an object)

- This is indicated by a normal that points away from the camera
  - Angle can be computed using dot product
    \[ n \cdot v \leq 0 \]
    \[ n \cdot v \geq 0 \]
  - Easier: Orientation of a triangle with respect to the camera (counterclockwise vs. clockwise). Orientation is defined by the order in which you specify vertices that form a triangle.
  - In OpenGL: signed area of triangle (cross-product) indicates front or back facing triangles
Transparency and Depth

- Transparency is typically modeled using **alpha-blending**
- Final color contains contributions of multiple fragments

\[
RGB_{final} = (1 - A_{in}) \cdot RGB_{existing} + A_{in} \cdot RGB_{in}
\]
Transparency and Depth

- If we render front-to-back with depth-testing, fragments at the back cannot contribute.
- If we render front-to-back (or in arbitrary order) without depth-testing, order of color computation is reversed (incorrect).
Transparency and Depth

- **Solution: Render back-to-front**
  - Geometry needs to be resorted (possibly every frame)
  - Depth-sort not always possible
  - Order-independent blending modes possible (weighted averaging)
  - If we combine transparent and opaque objects, simply render opaque ones first, updating the z-buffer

- Other (multi-pass) solutions exist
  - Depth-Peeling