• How is it?
• Remote making its first feeble steps?

• Come to labs
• Learn how to use debugger console
LAST TIME

• What does the vertex shader do?
• - fragment shader?
• How to pass a single value from JS to Vertex Shader?
PIPELINE: MORE DETAILS

Vertices and attributes → Vertex Shader → Vertex Post-Processing → Rasterization → Framebuffer

- - - → Fragment Shader → Per-Sample Operations →
PIPELINE: MORE DETAILS

**Vertex Shader**
- Modelview transform
- Per-vertex attributes

**Vertex Post-Processing**
- Viewport transform
- Clipping

**Rasterization**
- Scan conversion
- Interpolation

**Fragment Shader**
- Texturing/...
- Lighting/shading

**Per-Sample Operations**
- Depth test
- Blending

**Framebuffer**
SHAPES: REPRESENTATION OPTIONS

• Volumetric – Boolean algebra with volumetric primitives
  • Spheres, cones, cylinders, tori, ...

• Boundary representation – union of surface patches
  • Single basic primitive – Triangle Mesh
  • Higher order surface/curve primitives
SHAPES - CURVES/SURFACES

• Mathematical representations:
  • Explicit functions
  • Parametric functions
  • Implicit functions
SHAPES: EXPLICIT FUNCTIONS

• Curves:
  • $y$ is a function of $x$:
  • Only works in 2D

• Surfaces:
  • $z$ is a function of $x$ and $y$:
  • Cannot define arbitrary shapes in 3D

\[
y := \sin(x) \quad z := \sin(x) + \cos(y)
\]
SHAPES: PARAMETRIC FUNCTIONS

• Curves:
  • 2D: x and y are functions of a parameter value t
  • 3D: x, y, and z are functions of a parameter value t

\[
C(t) := \begin{pmatrix}
\cos(t) \\
\sin(t) \\
t
\end{pmatrix}
\]
Surfaces:

- Surface $S$ is defined as a function of parameter values $s, t$
- Names of parameters can be different to match intuition:

$$S(\phi, \theta) := \begin{pmatrix} \cos(\phi) \cos(\theta) \\ \sin(\phi) \cos(\theta) \\ \sin(\theta) \end{pmatrix}$$
SHAPES: IMPLICIT

- Surface (3D) or Curve (2D) defined by zero set (roots) of function
  - E.g:

  \[ S(x, y, z) : x^2 + y^2 + z^2 - 1 = 0 \]
SHAPES: TRIANGLE MESHES

• Triangle = 3 vertices

• Mesh = \{\text{vertices, triangles}\}

• Example
PIPELINE: MORE DETAILS

**Vertex Shader**
- Modelview transform
- Per-vertex attributes

**Vertex Post-Processing**
- Viewport transform
- Clipping

**Rasterization**
- Scan conversion
- Interpolation

**Fragment Shader**
- Texturing/...
- Lighting/shading

**Per-Sample Operations**
- Depth test
- Blending

**Framebuffer**
MODELING AND VIEWING TRANSFORMATIONS

• Placing objects - Modeling transformations
  • Map points from object coordinate system to world coordinate system

• Looking from the camera - Viewing transformation
  • Map points from world coordinate system to camera (or eye) coordinate system
MODELING TRANSFORMATIONS: OBJECT PLACEMENT
VIEWING TRANSFORMATION: LOOKING FROM A CAMERA
MODELING & VIEWING TRANSFORMATIONS

- Other transformations (not handled by rendering pipeline):
  - Freeform deformation
MODELING & VIEWING TRANSFORMATION

• Linear transformations
  • Rotations, scaling, shearing
  • Can be expressed as 3x3 matrix
  • E.g. scaling (non uniform):

\[
\begin{pmatrix}
  x' \\
  y' \\
  z'
\end{pmatrix} =
\begin{pmatrix}
  2 & 0 & 0 \\
  0 & 3 & 0 \\
  0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
  x \\
  y \\
  z
\end{pmatrix}
\]
• Affine transformations
  • Linear transformations + translations
  • Can be expressed as 3x3 matrix + 3 vector
  • E.g. scale+ translation:

\[
\begin{pmatrix}
x' \\
y' \\
z'
\end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}
\]

• Another representation: 4x4 homogeneous matrix
MATRICES

• Object coordinates -> World coordinates
  • Model Matrix
  • One per object

• World coordinates -> Camera coordinates
  • View Matrix
  • One per camera
PIPELINE: MORE DETAILS

Vertex Shader
- Modelview transform
- Per-vertex attributes

Vertex Post-Processing
- Viewport transform
- Clipping

Rasterization
- Scan conversion
- Interpolation

Fragment Shader
- Texturing/...
- Lighting/shading

Per-Sample Operations
- Depth test
- Blending

Framebuffer
PERSPECTIVE TRANSFORMATION

• Purpose:
  • Project 3D geometry to 2D image plane
  • Simulates a camera

• Camera model:
  • Pinhole camera (single view point)
  • More complex camera models exist, but are less common in CG
PERSPECTIVE PROJECTION
In computer graphics:
  - Image plane conceptually in front of center of projection

Perspective transformation is **one of** projective transformations
  - Linear & affine transformations also belong to this class
  - All projective transformations can be expressed as 4x4 matrix operations
PIPELINE: MORE DETAILS

- **Vertices and attributes**
  - Vertex Shader
    - Modelview transform
    - Per-vertex attributes
  - Rasterization
    - Scan conversion
    - Interpolation
  - Per-Sample Operations
    - Depth test
    - Blending
  - Vertex Post-Processing
    - Viewport transform
    - Clipping
  - Fragment Shader
    - Texturing/...
    - Lighting/shading
  - Framebuffer
CLIPPING

• Removing invisible geometry
  • Geometry outside viewing frustum
  • Plus too far or too near one

• Optimization
SCAN CONVERSION/RASTERIZATION

• Convert continuous 2D geometry to discrete
• Raster display – discrete grid of elements
• Terminology
  • **Screen Space:** Discrete 2D Cartesian coordinate system of the screen pixels
SCAN CONVERSION
SCAN CONVERSION
SCAN CONVERSION

• Problem:
  • Line is infinitely thin, but image has finite resolution
  • Results in steps rather than a smooth line
    • Jaggies
    • Aliasing
  • One of the fundamental problems in computer graphics
SCAN CONVERSION
COLOR INTERPOLATION

Linearly interpolate per-pixel color from vertex color values. Treat every channel of RGB color separately.
COLOR INTERPOLATION

• Example:
PIPELINE: MORE DETAILS

Vertices and attributes

**Vertex Shader**
- Modelview transform
- Per-vertex attributes

**Vertex Post-Processing**
- Viewport transform
- Clipping

**Rasterization**
- Scan conversion
- Interpolation

**Fragment Shader**
- Texturing/...
- Lighting/shading

**Per-Sample Operations**
- Depth test
- Blending

**Framebuffer**
TEXTURING

\[(s_0, t_0), (s_1, t_1)\]

\[(s_2, t_2)\]
TEXTURING

$$(s_0, t_0), (s_1, t_1), (s_2, t_2)$$
TEXTURE MAPPING
DISPLACEMENT MAPPING
TEXTURING

• Issues:
  • Computing 3D/2D map (low distortion)
  • How to map pixel from texture (texels) to screen pixels
    • Texture can appear widely distorted in rendering
    • Magnification / minification of textures
  • Filtering of textures
  • Preventing aliasing (anti-aliasing)
PIPELINE: MORE DETAILS

- **Vertices and attributes**
  - Modelview transform
  - Per-vertex attributes

- **Rasterization**
  - Scan conversion
  - Interpolation

- **Fragment Shader**
  - Texturing/...
  - Lighting/shading

- **Vertex Shader**
- **Vertex Post-Processing**
  - Viewport transform
  - Clipping

- **Per-Sample Operations**
  - Depth test
  - Blending

- **Framebuffer**
LIGHTING
PIPELINE: MORE DETAILS

1. **Vertex Shader**
   - Modelview transform
   - Per-vertex attributes

2. **Vertex Post-Processing**
   - Viewport transform
   - Clipping

3. **Rasterization**
   - Scan conversion
   - Interpolation

4. **Fragment Shader**
   - Texturing/...
   - Lighting/shading

5. **Per-Sample Operations**
   - Depth test
   - Blending

6. **Framebuffer**
WITHOUT HIDDEN LINE REMOVAL
HIDDEN LINE REMOVAL
HIDDEN SURFACE REMOVAL
DEPTH TEST /HIDDEN SURFACE REMOVAL

• Remove invisible geometry
  • Parts that are hidden behind other geometry

• Possible Implementations:
  • Pixel level decision
    • Depth buffer
  • Object space decision
    • E.g. intersection order for ray tracing
BLENDING

• Blending:
  • Fragments -> Pixels
  • Draw from farthest to nearest
  • No blending – replace previous color
  • Blending: combine new & old values with some arithmetic operations

• Frame Buffer: video memory on graphics board that holds resulting image & used to display it
REFLECTION/SHADOWS