A1

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

• Come to labs
• Learn how to use debugger console
THEORY ASSIGNMENT 1

• Math recap
• Due in a week in class (Sep 23rd)

LAST TIME

• What does the vertex shader do?
LAST TIME

• What does the vertex shader do?
• Fragment shader?

LAST TIME

• What does the vertex shader do?
• Fragment shader?
• How to pass some value from JS to Vertex Shader?
**LAST TIME**

- What does the vertex shader do?
- Fragment shader?
- How to pass a single value from JS to Vertex Shader?

**VERTEX SHADER**

- VS is run for each vertex SEPARATELY

Object coordinates -> WORLD coordinates -> **VIEW coordinates**
**FRAGMENT SHADER**

- **Uniform variables**
- **Varying variables**
- **Fragment shader**
- **Screen color**
- **Frame buffer**

**CONCEPTS**

- **uniform**  
  - same for all vertices
- **varying**  
  - computed per vertex, automatically interpolated for fragments
- **attribute**  
  - some values per vertex
  - available only in Vertex Shader

---

**JS + Three.JS → Vertex Shader → Fragment Shader**
PIPELINE: MORE DETAILS

Vertices and attributes → Vertex Shader → Vertex Post-Processing → Rasterization → Fragment Shader → Per-Sample Operations → 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:
  - $y := \sin(x)$
  - $y$ is a function of $x$:
  - Only works in 2D

- Surfaces:
  - $z := \sin(x) + \cos(y)$
  - $z$ is a function of $x$ and $y$:
  - Cannot define arbitrary shapes in 3D

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 $t$

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

• 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 = \{vertices, triangles\}

• 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
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):

\[
\begin{align*}
\begin{pmatrix}
    x' \\
    y' \\
    z'
\end{pmatrix} &=
\begin{pmatrix}
    2 & 0 & 0 \\
    0 & 3 & 0 \\
    0 & 0 & 1
\end{pmatrix} \cdot
\begin{pmatrix}
    x \\
    y \\
    z
\end{pmatrix}
\end{align*}
\]

- Affine transformations
  - Linear transformations + translations
  - Can be expressed as 3x3 matrix + 3 vector
  - E.g. scale+ translation:

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

- 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

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
PERSPECTIVE TRANSFORMATION

• 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

PIPELINE: MORE DETAILS

Vertices and attributes

Vertex Shader
- Modelview transform
- Per-vertex attributes

Rasterization
- Scan conversion
- Interpolation

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

Per-Sample Operations
- Depth test
- Blending

Vertex Post-Processing
- Viewport transform
- Clipping

Framebuffer
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

• 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

• Linearly interpolate per-pixel color from vertex color values
  • Treat every channel of RGB color separately

COLOR INTERPOLATION

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

• Example:

![Example Diagram]

PIPELINE: MORE DETAILS

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

COMPLEX LIGHTING AND SHADING
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

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

PIPELINE: MORE DETAILS

Vertices and attributes → Vertex Shader
  - Modelview transform
  - Per-vertex attributes
  → Rasterization
    - Scan conversion
    - Interpolation
  → Fragment Shader
    - Texturing/...
    - Lighting/shading
  → Vertex Post-Processing
    - Viewport transform
    - Clipping
  → Framebuffer

Per-Sample Operations
  - Depth test
  - Blending
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
COORDINATE SYSTEMS

• Coordinate system = Origin + Basis Vectors
COORDINATE SYSTEMS

\[ P = O + x \vec{i} + y \vec{j} \]

equivalent: \( P = (x, y) \)

COORDINATE SYSTEMS

\[ P = O + x \vec{i} + y \vec{j} \]

\( F_1 \), \( F_2 \), \( F_3 \)
COORDINATE SYSTEMS

\[ P = O + x\vec{i} + y\vec{j} \]

- \( F_1 \) \( P(3, -1) \)
- \( F_2 \)
- \( F_3 \)

COORDINATE SYSTEMS

\[ P = O + x\vec{i} + y\vec{j} \]

- \( F_1 \) \( P(3, -1) \)
- \( F_2 \) \( P(-1.5, 2) \)
- \( F_3 \)
COORDINATE SYSTEMS

\[ P = O + x\mathbf{i} + y\mathbf{j} \]

- \( F_1 \) P(3,-1)
- \( F_2 \) P(-1.5,2)
- \( F_3 \) P(1,2)

BONUS (WARM-UP)

- For a given vector and a coordinate frame, are vector's coordinates unique? Why? Are they always defined?