Correction: Vector-Vector Multiplication
- multiply: vector * vector = scalar
- dot product, aka inner product
\[
\begin{bmatrix}
    u_1 \\
    u_2 \\
    u_3
\end{bmatrix}
\cdot
\begin{bmatrix}
    v_1 \\
    v_2 \\
    v_3
\end{bmatrix}
= (u_1 \cdot v_1) + (u_2 \cdot v_2) + (u_3 \cdot v_3)
\]
- geometric interpretation
  - lengths, angles
  - can find angle between two vectors

Correction: Dot Product Example
\[
\begin{bmatrix}
    6 \\
    1 \\
    2
\end{bmatrix}
\cdot
\begin{bmatrix}
    1 \\
    7 \\
    3
\end{bmatrix}
= (6 \cdot 1) + (1 \cdot 7) + (2 \cdot 3) = 6 + 7 + 6 = 19
\]

More: Working with Frames
- p = \(0 + t(1)\)
- p = \(3, -1\)
- p = \(-1.5, 2\)
- p = \((1, 2)\)

Lines
- slope-intercept form
  - \(y = mx + b\)
- implicit form
  - \(y - mx - b = 0\)
  - \(Ax + By + C = 0\)
  - \(f(x, y) = 0\)

Implicit Functions
- find where function is 0
- plug in \((x, y)\)
  - \(0\): on line
  - \(> 0\): inside
  - \(< 0\): outside
- analogy: terrain
  - sea level: \(t=0\)
  - altitude: function value
  - topo map: equal-value contours (level sets)

Implicit Circles
- \(f(x, y) = (x-x_0)^2 + (y-y_0)^2 - r^2\)
- circle is points \((x, y)\) where \(f(x, y) = 0\)
- \(p = (x, y), c = (x_c, y_c): (p - c) \cdot (p - c) - r^2 = 0\)
- points p on circle have property that vector from c to p dotted with itself has value \(r^2\)
- \(p \cdot p - r^2 = 0\)
- points points p on the circle have property that squared distance from c to p is \(r^2\)
- \(p \cdot p - r^2 = 0\)
- points p on circle are those a distance r from center point c

Parameterized Curves
- parameter: index that changes continuously
- \((x(t), y(t))\): point on curve
- \(t\): parameter
- vector form
  - \(\mathbf{p} = f(t)\)

2D Parameteric Lines
- \(\mathbf{p}(t) = \mathbf{p}_1 + t(\mathbf{p}_2 - \mathbf{p}_1)\)
- start at point \(\mathbf{p}_0\), go towards \(\mathbf{p}_1\) according to paramater \(t\)
- \(t=0\): \(\mathbf{p}_0\)
- \(t=1\): \(\mathbf{p}_1\)
Matrix-Matrix Multiplication

- add: matrix + matrix = matrix
  \[
  \begin{bmatrix}
  m_{11} & m_{12} \\
  m_{21} & m_{22}
  \end{bmatrix}
  +
  \begin{bmatrix}
  n_{11} & n_{12} \\
  n_{21} & n_{22}
  \end{bmatrix}
  =
  \begin{bmatrix}
  m_{11} + n_{11} & m_{12} + n_{12} \\
  m_{21} + n_{21} & m_{22} + n_{22}
  \end{bmatrix}
  \]

- example
  \[
  \begin{bmatrix}
  1 & 3 \\
  2 & 4
  \end{bmatrix}
  +
  \begin{bmatrix}
  -2 & 5 \\
  7 & 1
  \end{bmatrix}
  =
  \begin{bmatrix}
  1 + (-2) & 3 + 5 \\
  2 + 7 & 4 + 1
  \end{bmatrix}
  =
  \begin{bmatrix}
  -1 & 8 \\
  9 & 5
  \end{bmatrix}
  \]

Scalar-Matrix Multiplication

- multiply: scalar * matrix = matrix
  \[
  p \cdot \begin{bmatrix}
  m_{11} & m_{12} \\
  m_{21} & m_{22}
  \end{bmatrix}
  =
  \begin{bmatrix}
  pm_{11} & pm_{12} \\
  pm_{21} & pm_{22}
  \end{bmatrix}
  \]

- example
  \[
  2 \cdot \begin{bmatrix}
  1 & 2 \\
  4 & 5
  \end{bmatrix}
  =
  \begin{bmatrix}
  2 \cdot 1 & 2 \cdot 2 \\
  2 \cdot 4 & 2 \cdot 5
  \end{bmatrix}
  =
  \begin{bmatrix}
  2 & 4 \\
  8 & 10
  \end{bmatrix}
  \]

Matrix-Matrix Addition

- add: matrix + matrix = matrix
  \[
  \begin{bmatrix}
  m_{11} & m_{12} \\
  m_{21} & m_{22}
  \end{bmatrix}
  +
  \begin{bmatrix}
  n_{11} & n_{12} \\
  n_{21} & n_{22}
  \end{bmatrix}
  =
  \begin{bmatrix}
  m_{11} + n_{11} & m_{12} + n_{12} \\
  m_{21} + n_{21} & m_{22} + n_{22}
  \end{bmatrix}
  \]

Matrices and Linear Systems

- linear system of n equations, n unknowns
  \[
  \begin{bmatrix}
  a & b & c \\
  d & e & f \\
  g & h & i
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y \\
  z
  \end{bmatrix}
  =
  \begin{bmatrix}
  j \\
  k \\
  l
  \end{bmatrix}
  \]

- undefined
  \[
  \begin{bmatrix}
  a & b & c \\
  e & f & g \\
  o & p & q
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y \\
  z
  \end{bmatrix}
  =
  \begin{bmatrix}
  h \\
  i \\
  k
  \end{bmatrix}
  \]

Rendering Pipeline

- what is the pipeline?
- abstract model for sequence of operations to transform geometric model into digital image
- abstraction of the way graphics hardware works
- underlying model for application programming interfaces (APIs) that allow programming of graphics hardware
- OpenGL
- Direct 3D
- actual implementation details of rendering pipeline will vary

Rendering

- goal
  - transform computer models into images
  - may or may not be photo-realistic
- interactive rendering
  - fast, but limited quality
  - roughly follows a fixed patterns of operations
  - rendering pipeline
- offline rendering
  - ray tracing
  - global illumination

Rendering

- tasks that need to be performed
  - in no particular order:
    - project all 3D geometry onto the image plane
    - geometric transformations
    - determine which primitives or parts of primitives are visible
    - hidden surface removal
    - determine which pixels a geometric primitive covers
    - scan conversion
    - compute the color of every visible surface point
      - lighting, shading, texture mapping

Rendering Pipeline

- Geometry Database
- Model/View Transform
- Lighting
- Perspective Transform
- Clipping
- Scan Conversion
- Texturing
- Depth Test
- Blending
- Framebuffer
Geometry Database
- geometry database
- application-specific data structure for holding geometric information
- depends on specific needs of application
  - triangle soup, points, mesh with connectivity information, curved surface

Model/View Transformation
- modeling transformation
- map all geometric objects from local coordinate system into world coordinates
- viewing transformation
- map all geometry from world coordinates into camera coordinates

Lighting
- compute brightness based on property of material and light position(s)
- computation is performed per-vertex

Clipping
- clipping
- removal of parts of the geometry that fall outside the visible screen or window region
- may require re-tessellation of geometry

Geometry Database
- geometry database
- application-specific data structure for holding geometric information
- depends on specific needs of application
  - triangle soup, points, mesh with connectivity information, curved surface

Pipeline Advantages
- modularity: logical separation of different components
- easy to parallelize
  - earlier stages can already work on new data while later stages still work with previous data
  - similar to pipelining in modern CPUs
  - but much more aggressive parallelization possible (special purpose hardware!)
  - important for hardware implementations
  - only local knowledge of the scene is necessary

Pipeline Disadvantages
- limited flexibility
- some algorithms would require different ordering of pipeline stages
- hard to achieve while still preserving compatibility
- only local knowledge of scene is available
- shadows, global illumination difficult

OpenGL
- API to graphics hardware
- based on IRIS_GL by SGI
- designed to exploit hardware optimized for display and manipulation of 3D graphics
- implemented on many different platforms
- low level, powerful flexible
- pipeline processing
- set state as needed

Graphics State
- set the state once, remains until overwritten
  - glColor3f(1.0, 0.0, 0.0) → set color to yellow
  - glSetClearColor(0.0, 0.0, 0.2) → dark blue bg
  - glEnable(LIGHT0) → turn on light
  - glEnable(GL_DEPTH_TEST) → hidden surf.

Depth Test
- depth test
- remove parts of geometry hidden behind other geometric objects
- perform on every individual fragment
- other approaches (later)

Perspective Transformation
- perspective transformation
- projecting the geometry onto the image plane
- projective transformations and model/view transformations can all be expressed with 4x4 matrix operations

Blending
- blending
- final image: write fragments to pixels
- draw from farthest to nearest
- no blending – replace previous color
- blending: combine new & old values with arithmetic operations

OpenGL (briefly)
- tell it how to interpret geometry
  - glBegin(“mode of geometric primitive”)
  - mode = GL_TRIANGLE, GL_POLYGON, etc.
- feed it vertices
  - glVertex3f(-1.0, 0.0, -1.0)
  - glVertex3f(1.0, 0.0, -1.0)
  - glVertex3f(0.0, 1.0, -1.0)
- tell it you’re done
  - glEnd()
Open GL: Geometric Primitives

void display()
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0, 1.0, 0.0);
    glBegin(GL_POLYGON);
    glVertex3f(0.25, 0.25, -0.5);
    glVertex3f(0.75, 0.25, -0.5);
    glVertex3f(0.75, 0.75, -0.5);
    glVertex3f(0.25, 0.75, -0.5);
    glEnd();
    glFlush();
}

• more OpenGL as course continues