Computer Graphics Seminar

MTAT.03.305

Fall 2020

Raimond Tunnel
Computer Graphics

- Graphical illusion via the computer
- Displaying something meaningful (incl art)
Math

- Computers are good at... computing.
- To do computer graphics, we need math for the computer to compute.
- Geometry, algebra, calculus.

\[
\begin{bmatrix}
a & b & c & d \\
e & f & g & h \\
i & j & k & l \\
0 & 0 & 0 & 1 \\
\end{bmatrix} \cdot \begin{bmatrix}
x \\
y \\
z \\
1 \\
\end{bmatrix} = \begin{bmatrix}
ax + by + cz + d \\
ex + fy + gz + h \\
ix + jy + kz + l \\
1 \\
\end{bmatrix}
\]
Math

For creating and manipulating 3D objects we use:

- **Analytic geometry** – math about coordinate systems
- **Linear algebra** – math about vectors and spaces
Skills for Computer Graphics

- Mathematical understanding
  \[
  \begin{pmatrix}
  a & b \\
  c & d
  \end{pmatrix}
  \begin{pmatrix}
  x \\
  y
  \end{pmatrix}
  =
  \begin{pmatrix}
  ax + by \\
  cx + dy
  \end{pmatrix}
  \]

- Geometrical (spatial) thinking

- Programming
  ```
  GLuint vaoHandle;
  glGenVertexArrays(1, &vaoHandle);
  glBindVertexArray(vaoHandle);
  ```

- Visual creativity & aesthetics
The Standard Graphics Pipeline

1. Define geometry and transformations
2. Apply geometry and transformations
3. Culling & Clipping
4. Rasterization
5. Fragment shading
6. Visibility tests & Blending
Point

- Simplest geometry primitive
- In homogeneous coordinates: $(x, y, z, w), w \neq 0$

- Represents a point $(x/w, y/w, z/w)$
- Usually you can put $w = 1$ for points
- Actual division will be done by GPU later
Line (segment)

- Consists of:
  - 2 endpoints
  - *Infinite* number of points between
- Defined by the endpoints
- Interpolated and rasterized in the GPU
Line (segment)

• Consists of:
  • 2 endpoints
  • *Infinite* number of points between

• Defined by the endpoints

• Interpolated and rasterized in the GPU
Line (segment)

- Consists of:
  - 2 endpoints
  - *Infinite* number of points between
- Defined by the endpoints
- Interpolated and rasterized in the GPU
Triangle

- Consists of:
  - 3 points called vertices
  - 3 lines called edges
  - 1 face

- Defined by 3 vertices

- Face interpolated and rasterized in the GPU

- Counter-clockwise order defines the front face
Triangle

- Consists of:
  - 3 points called vertices
  - 3 lines called edges
  - 1 face

- Defined by 3 vertices

- Face interpolated and rasterized in the GPU

- Counter-clockwise order defines the **front face**
Why triangles?

• They are in many ways the simplest polygons
  • 3 different points always form a plane
  • Easy to rasterize (fill the face with pixels)
  • Every other polygon can be converted to triangles
Why triangles?

- They are in many ways the simplest polygons
  - 3 different points always form a plane
  - Easy to rasterize (fill the face with pixels)
  - Every other polygon can be converted to triangles

- OpenGL used to support other polygons too
  - Must have been:
    - **Simple** – No edges intersect each other
    - **Convex** – All points between any two inner points are inner points
Examples of polygons
OpenGL < 3.1 primitives

Figures 2-7: Geometric Primitive Types

OpenGL Programming Guide 7th edition, p49
After OpenGL 3.1

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>OpenGL Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points</td>
<td>GL_POINTS</td>
</tr>
<tr>
<td>Lines</td>
<td>GL_LINES</td>
</tr>
<tr>
<td>Line Strips</td>
<td>GL_LINE_STRIP</td>
</tr>
<tr>
<td>Line Loops</td>
<td>GL_LINE_LOOP</td>
</tr>
<tr>
<td>Independent Triangles</td>
<td>GL_TRIANGLES</td>
</tr>
<tr>
<td>Triangle Strips</td>
<td>GL_TRIANGLE_STRIP</td>
</tr>
<tr>
<td>Triangle Fans</td>
<td>GL_TRIANGLE_FAN</td>
</tr>
</tbody>
</table>

**Figure 3.1**  Vertex layout for a triangle strip

**Figure 3.2**  Vertex layout for a triangle fan
In the beginning there were points

- We can now define our geometric objects!
In the beginning there were points

- We can now define our geometric objects!
- We want to move our objects!

World's $(0, 0, 0)$
Transformations

- **Linear transformations**
  - Scaling, reflection
  - Rotation
  - Shearing

- **Affine transformations**
  - Translation (moving / shifting)

- **Projection transformations**
  - Perspective
  - Orthographic

Homogeneous coordinates are needed here...

...and for the perspective projection
Transformations

- Every transformation is a function
- As you have learned from algebra, **all linear functions can be represented as matrices**

\[
f(v) = \begin{pmatrix} 2 \cdot x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}
\]

\[v \in \mathbb{R}^3\]

Column-major format
Transformations

- Every transformation is a function
- As you have learned from algebra, all linear functions can be represented as matrices

\[
f(v) = \begin{pmatrix} 2 \cdot x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}
\]

Linear function, which increases the first coordinate two times.

Column-major format

\[
v \in \mathbb{R}^3
\]

\[
v = \begin{pmatrix} x \\ y \\ z \end{pmatrix}
\]
Transformations

• Every transformation is a function

• As you have learned from algebra, all linear functions can be represented as matrices

\[
\begin{align*}
f(v) &= \begin{pmatrix} 2 \cdot x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \\
v \in \mathbb{R}^3 \\
v &= \begin{pmatrix} x \\ y \\ z \end{pmatrix}
\end{align*}
\]

Same function as a matrix

Column-major format
Transformations

- GPU-s are built for doing transformations with matrices on points (vertices).

\[ M \cdot v_0 \]
\[ M \cdot v_1 \]
\[ M \cdot v_2 \]

Vertex shader code
Transformations

• GPU-s are built for doing transformations with matrices on points (vertices).

• Linear transformations satisfy:

\[ f(a_1 x_1 + \ldots + a_n x_n) = a_1 f(x_1) + \ldots + a_n f(x_n) \]

We will not use homogeneous coordinates at the moment, but they will be back...
Linear Transformation

Scale
Scaling

- Multiplies the coordinates by a scalar factor.

$$\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

$$\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1.5 \\ 1.5 \end{pmatrix} = \begin{pmatrix} 3 \\ 1.5 \end{pmatrix}$$
Scaling

- Multiplies the coordinates by a scalar factor.
- Scales the standard basis vectors / axes.

\[
\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix} = e_0
\]

\[
\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = e_1
\]
Scaling

- In general we could scale each axis

\[
\begin{pmatrix}
  a_x & 0 & 0 \\
  0 & a_y & 0 \\
  0 & 0 & a_z
\end{pmatrix}
\]

\(a_x\) – x-axis scale factor

\(a_y\) – y-axis scale factor

\(a_z\) – z-axis scale factor

- If some factor is negative, this matrix will reflect the points from that axis. Thus we get reflection.

What happens to our triangles when an odd number of factors are negative?
Linear Transformation

Shear
Shearing

- Remember it for translations later.
- Tilts only one axis.
- Squares become parallelograms.

\[
\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 2 \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \end{pmatrix}
\]

\[
\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}
\]

\[
\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}
\]
Shearing

- **Shear-y**, we tilt parallel to y-axis by angle $\varphi$ counter-clockwise

\[
\begin{pmatrix}
1 & 0 \\
\tan(\varphi) & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y
\end{pmatrix} =
\begin{pmatrix}
x \\
y + \tan(\varphi) \cdot x
\end{pmatrix}
\]

- **Shear-x**, we tilt parallel to x-axis by angle $\varphi$ clockwise

\[
\begin{pmatrix}
1 & \tan(\varphi) \\
0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y
\end{pmatrix} =
\begin{pmatrix}
x + \tan(\varphi) \cdot y \\
y
\end{pmatrix}
\]
Linear Transformation

Rotation
Rotation

- Shearing moved only one axis
- Also changed the size of the basis vector
- Can we do better?

Did you notice that the columns of the transformation matrix show the coordinates of the new basis vectors?
Rotation

\[ e'_0 = (|a|, |b|) = (\cos(\alpha), \sin(\alpha)) \]
\[ e'_1 = (|a'|, |b'|) = (-\sin(\alpha), \cos(\alpha)) \]

\[ \cos(\alpha) = \frac{|a|}{|e'_0|} = \frac{|a|}{1} = |a| \]
So if we rotate by \( \alpha \) in counter-clockwise order in 2D, the transformation matrix is:

\[
\begin{pmatrix}
\cos(\alpha) & -\sin(\alpha) \\
\sin(\alpha) & \cos(\alpha)
\end{pmatrix}
\]

In 3D we can do rotations in each plane (xy, xz, yz), so there can be 3 different matrices.
Rotation

- To do a rotation around an arbitrary axis, we can:
  - Rotate that axis to be the x-axis
  - Rotate around the new x-axis
  - Invert the first rotations (move the old x-axis back)

- OpenGL provides a command for rotating around a given axis.

- Generally quaternions are used for rotations.

  Quaternions are elements of a number system that extend the complex numbers...
Do we have everything now?

- We can scale, shear and rotate our geometry around the origin...

What if we have an object not centered in the origin?
Affine Transformation

Translation
Translation

- Imagine that a 1D world is located at $y=1$ line in 2D space.

- Notice that all the points are in the form: $(x, 1)$
Translation

• Imagine that a 1D world is located at y=1 line in 2D space.

• Notice that all the points are in the form: (x, 1)
Translation

- Do a shear-x(45°) operation on the 2D world!

\[ \tan(45°) = 1 \]

- Everything has now moved 1 unit in x to the right from the original position.
Translation

- What if we do shear-x(63.4°)?

- We can do translation (movement)!

\[
\tan(45^\circ) = 1 \\
\tan(63.4^\circ) = 2
\]
Translation

- When we represent our points in one dimension higher space, where the extra coordinate is 1, we get to the **homogeneous** space.

\[
\begin{pmatrix}
1 & x_t \\
0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
1
\end{pmatrix}
= 
\begin{pmatrix}
x + x_t \\
1
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 0 & x_t \\
0 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
1
\end{pmatrix}
= 
\begin{pmatrix}
x + x_t \\
y + y_t \\
1
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 0 & 0 & x_t \\
0 & 1 & 0 & y_t \\
0 & 0 & 1 & z_t \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z \\
1
\end{pmatrix}
= 
\begin{pmatrix}
x + x_t \\
y + y_t \\
z + z_t \\
1
\end{pmatrix}
\]
Transformations

- This together gives us a very good **toolset** to transform our geometry as we wish.

![Augmented transformation matrix!](image)

\[
\begin{pmatrix}
  a & b & c & x_t \\
  d & e & f & y_t \\
  g & h & i & z_t \\
  0 & 0 & 0 & 1
\end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix}
  ax + by + cz + x_t \\
  dx + ey + fz + y_t \\
  gx + hy + iz + z_t \\
  1
\end{pmatrix}
\]
Multiple transformations

- Everything starts from the origin!
- To apply multiple transformations, just multiply matrices.
Multiple transformations

Our initial geometry defined by vertices: (-1, -1), (1, -1), (1, 1), (-1, 1)
Multiple transformations

\[
\begin{pmatrix}
\cos(\theta) & -\sin(\theta) & 0 \\
\sin(\theta) & \cos(\theta) & 0 \\
0 & 0 & 1
\end{pmatrix}
\]
Multiple transformations

\[
\begin{pmatrix}
1 & 0 & 4 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{pmatrix}
\]
Multiple transformations

- Combine the transformations to a single matrix.

\[
\begin{pmatrix}
1 & 0 & 4 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{pmatrix} \cdot \begin{pmatrix}
\cos(45°) & -\sin(45°) & 0 \\
\sin(45°) & \cos(45°) & 0 \\
0 & 0 & 1 \\
\end{pmatrix} = \begin{pmatrix}
\cos(45°) & -\sin(45°) & 4 \\
\sin(45°) & \cos(45°) & 0 \\
0 & 0 & 1 \\
\end{pmatrix}
\]

- This works for combining different affine transformations, but the result is hard to read...

- Order of transformations / matrices is important!

- http://cgdemos.tume-maailm.pri.ee
Now You Know

Data

Vertex transformations

OpenGL

Vertex shader

Culling & Clipping

Rasterization

Fragment shading

Visibility tests & Blending

\[
M = M_1 \cdot M_2 \cdot M_3 \cdot \ldots
\]

\[
P \cdot V \cdot M \cdot v = \begin{pmatrix} v_x, & v_y, & v_z \\ v_w, & v_w, & v_w \end{pmatrix}
\]
Next time...

- What is color in computer graphics?
- How to color our rasterized pixels?
- Light calculations.