Computer Graphics

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

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

\[
\begin{bmatrix}
ad \\
fh \ \\
ijk \ \\
0001
\end{bmatrix}
\begin{bmatrix}
xx \\
yy \\
z \\
1
\end{bmatrix}
= 
\begin{bmatrix}
ax + by + cz + d \\
ex + fy + gz + h \\
ix + jy + kz + l \\
1
\end{bmatrix}
\]
Point

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

\((x, y, z, 1)\)

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

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

\[
(x_1, y_1, z_1, 1) \quad \text{and} \quad (x_2, y_2, z_2, 1)
\]
Triangle

- Consists of:
  - 3 points called vertices
  - 3 lines called edges
  - 1 face
- Defined by 3 vertices
- Lines and face will be interpolated in GPU
- Counter-clockwise order defines 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 points)
  - 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 points are inner points
Examples of polygons
OpenGL < 3.1 primitives

Figure 2-7   Geometric Primitive Types

OpenGL Programming Guide 7th edition, p49
After OpenGL 3.1

Table 3.1   OpenGL Primitive Mode Tokens

<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

- Now that we can define our geometric objects, what next?
- We want to move our objects!

Luckily GPU will do this work for us.
Transformations

- It turns out that homogeneous coordinates allows us to easily do:
  - Linear transformations
    - Scaling, reflection
    - Rotation
    - Shearing
  - Affine transformations
    - Translation (shifting)
  - Projection transformations
    - Perspective
    - Orthographic

Actually these we could do without homogeneous coordinates...

This too...
Transformations

- Every transformation is a function
- As you remember from Algebra, linear functions can be represented as matrices

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

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

\[ v = (x, y, z) \]

You should transpose the result later
Transformations

- GPU-s are built for doing transformations with matrices.
- Remember, computers are made for computing.
- Let's look at some linear transformations...

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

We don't use homogeneous coordinates at the moment, don't worry, they'll be back...
Scaling

- Redefines the basis vectors as some multiple of the previous basis vectors.

\[
\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 2 \cdot 1.5 \\ 0 \cdot 1.5 \end{pmatrix} = \begin{pmatrix} 3 \\ 1.5 \end{pmatrix}
\]
Scaling

• In general we could scale each axis

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

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

- Not much used by itself, but remember it for translations later.
- Tilts only one axis.
- Squares become parallelograms.

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

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

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

- Shearing rotated 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

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

\[
\cos(\alpha) = \frac{|a|}{|e'_0|} = \frac{|a|}{1} = |a|
\]
Rotation

• 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.
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.

- Sometimes 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, share and rotate our geometry around the origin...

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

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

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

• What happens if we do shear-x(45°) operation on the 2D world?

• Everything in our world has moved magically one x-coordinate to the right...

\[ \tan(45°) = 1 \]
Translation

• What if we do shear-x(63.4°)?

\[ \tan(63.4°) = 2 \]

• Everything has now moved 2 x-coordinates to the right from the original position

• We can do translation / shift!
Translation

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

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

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

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

\[
\begin{pmatrix}
a & b & c & x_t \\
d & e & f & y_t \\
g & h & i & z_t \\
0 & 0 & 0 & 1
\end{pmatrix}
\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}

Used for perspective projection...
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

\[
\begin{pmatrix}
\cos(45^\circ) & -\sin(45^\circ) & 0 \\
\sin(45^\circ) & \cos(45^\circ) & 0 \\
0 & 0 & 1
\end{pmatrix}
\]
Multiple transformations

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

• We can combine the transformations to a single matrix.

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

• This also works for combining different linear transformations, but the resulting matrix isn't that clear...

• Order of transformations / matrices is important!

• http://cgdemos.tume-maailm.pri.ee
Now we know how it's supposed to go...
OpenGL

- GPU API / middleware
- Set of commands that program can give to GPU
- Supported in many languages

Vertices a, b, c
Transformations A, B, C
Draw!
WebGL

- GPU API in JavaScript
- Supported by major browsers
- THREE.js – Higher level library to ease your coding: http://threejs.org/docs/

Vertices a, b, c
Transformations A, B, C
Draw!
Forward is backward

• Actually you need to tell GPU commands in this order:
  • Transformations
  • Vertices

```c
glTranslatef(0.0, 0.0, -1.0);
glRotatef(60, 1.0, 0.0, 0.0);
glRotatef(-20, 0.0, 0.0, 1.0);
glBegin(GL_QUADS);
  glVertex3fv(...);
  glVertex3fv(...);
  glVertex3fv(...);
  glVertex3fv(...);
glEnd();
```

Prior to OpenGL 3
State machine

• GPU acts like a state machine

Prior to OpenGL 3
Multiple objects in the scene?

- Each object has its own geometry
- And its own transformations for that geometry
- OpenGL has a single ModelView matrix

Prior to OpenGL 3

```c
glTranslatef(0.0, 0.0, -1.0);
glRotatef(60, 1.0, 0.0, 0.0);
glRotatef(-20, 0.0, 0.0, 1.0);

glBegin(GL_QUADS);
    glVertex3fv(...);
    glVertex3fv(...);
    glVertex3fv(...);
    glVertex3fv(...);
glEnd();
```
Multiple objects in the scene

- OpenGL has a matrix stack
- We can push a copy to the stack (save)
- We can pop the top matrix from the stack (load)

Prior to OpenGL 3
Multiple objects in the scene

- More complex geometry for a single object

```gl
glPushMatrix();
glTranslatef(10.0, 0.0, -1.0);
drawPalm();
glPushMatrix();
glTranslatef(-1.0, 0.0, 0.0);
drawFingers();
glPopMatrix();
glPopMatrix();
```

Prior to OpenGL 3
Old and new OpenGL?

- Lot has changed from OpenGL 3.
- Everything old still works in compatibility mode.

Prior to OpenGL 3

- `glBegin(...)`
- `glVertex(...)`
- `glEnd(...)`
- `glTranslate(...)`
- `glRotate(...)`
- `glScale(...)`
- `glMaterial(...)`

OpenGL 3+

- Vertex Array Object (VAO)
- Vertex Buffer Object (VBO)
- Use other Matrix library
- Send your matrices to shaders
- Vertex Buffer Object

Good tutorials: [http://antongerdelan.net/opengl/](http://antongerdelan.net/opengl/)
Multiple objects in the scene

- Useful to think of the scene as a tree
Next time...

- Graphics pipeline in more detail
- How to define color for our geometry?
- Vertex and fragment shaders
- What else could be done?

Our geometry defined by 4 vertices
A parallelogram...