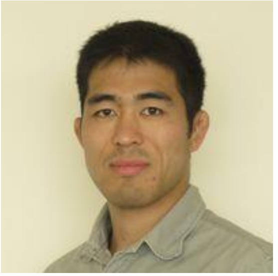
Introduction to Computer Graphics

April 7, 2016

Kenshi Takayama

Lecturers

- Kenshi Takayama (Assistant Prof., NII)
  - http://research.nii.ac.jp/~takayama/
  - takayama@nii.ac.jp



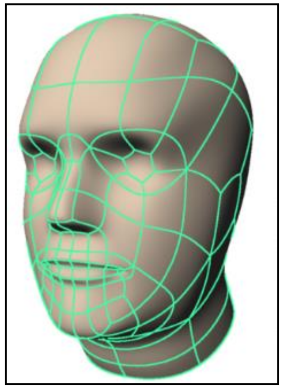
- Toshiya Hachisuka (Junior Associate Prof., U Tokyo)
  - http://www.ci.i.u-tokyo.ac.jp/~hachisuka/
  - thachisuka@siggraph.org

- Ryoichi Ando (Assistant Prof., NII)
  - https://scholar.google.com/citations?user=Ag3RwxUAAAAJ&hl=en

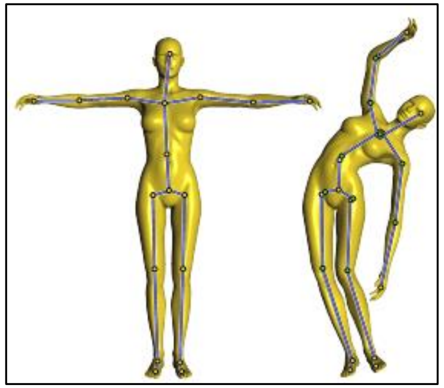
TA: Kazutaka Nakashima (Igarashi Lab)
http://n-taka.info/intro/
taka@ui.is.s.u-tokyo.ac.jp

Course overview

Modeling



Animation



Rendering

Image processing

- 2~3 lectures per topic, 12 lectures in total
- Rendering part by Prof. Hachisuka
- Fluid animation part by Prof. Ando

Grading

- Programming assignments only
  - No exam, no attendance check

- Two tyes of assignments: Basic & Advanced
  - Basic ➔ 1 assignent per topic (4 in total), very easy
  - Advanced ➔ For motivated students

- Deadline: The end of July

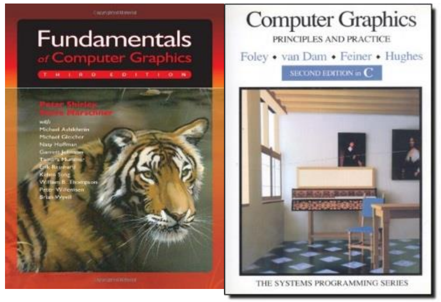
- Evaluation criteria
  - 1 assignemnt submitted ➔ **C** (bare minimum for the degree)
  - 4 assignments submitted ➔ **B** or higher
  - Distribution of **S** & **A** will be decided based on the quality/creativity of submissions and the overall balance in the class

- More details explained later

References

- Course website
  - http://research.nii.ac.jp/~takayama/teaching/utokyo-iscg-2016/

- Famous textbooks (not used in the class)
  - Fundamentals of Computer Graphics (9781568814698)
  - Computer Graphics: Principles and Practice in C (9780201848403)



Lecturers’ research topics

Coordinate transformations

Linear transformation

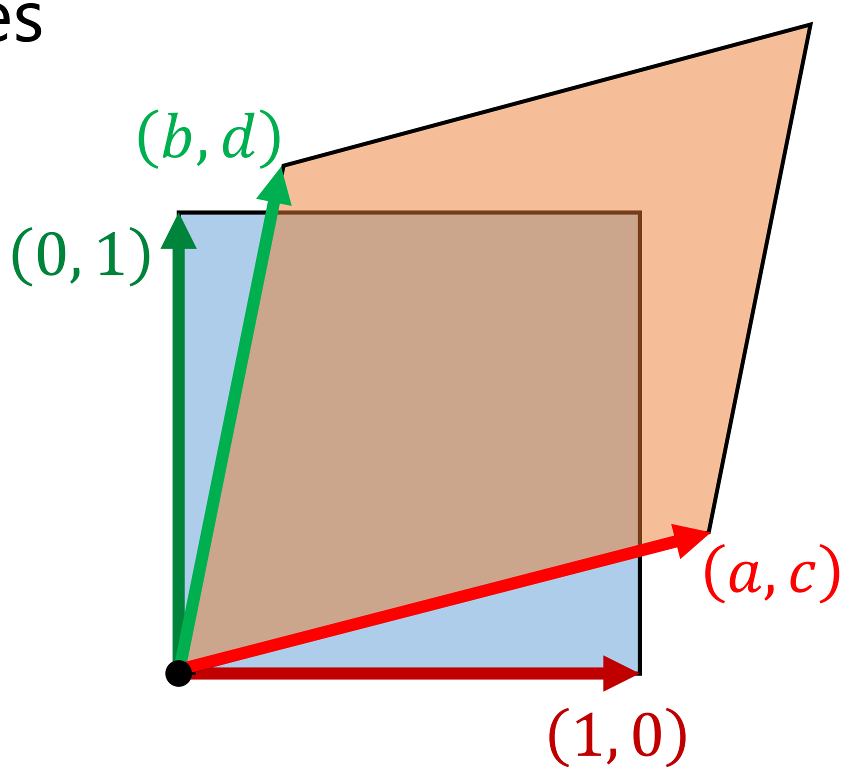
In 2D: $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$

In 3D: $\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$

- Intuition: Mapping of coordinate axes

$$\begin{bmatrix} a \\ c \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

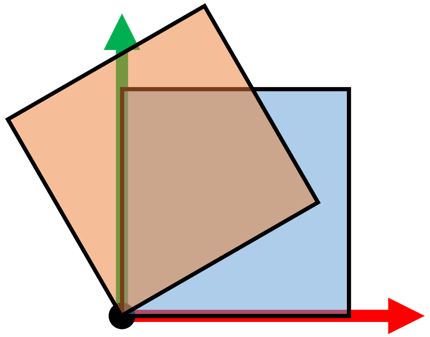
$$\begin{bmatrix} b \\ d \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$



- Origin stays put

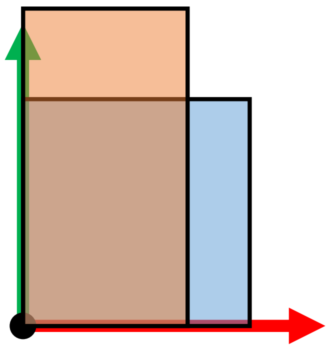
Special linear transformations

Rotation



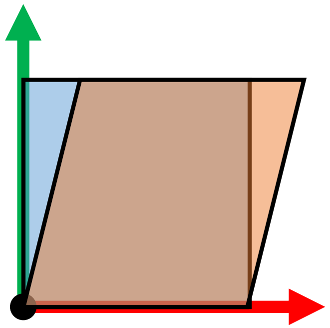
$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Scaling



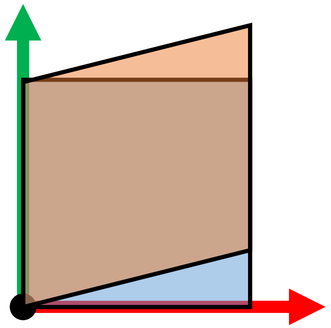
$$\begin{bmatrix} s_{\mathrm{x}} & 0 \\ 0 & s_{\mathrm{y}} \end{bmatrix}$$

Shearing (X dir.)



$$\begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}$$

Shearing (Y dir.)



$$\begin{bmatrix} 1 & 0 \\ k & 1 \end{bmatrix}$$

Linear transformation + translation = Affine transformation

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_{\mathrm{x}} \\ t_{\mathrm{y}} \end{bmatrix} \quad \Leftrightarrow \quad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & t_{\mathrm{x}} \\ c & d & t_{\mathrm{y}} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- Homogeneous coordinates: Use a 3D (4D) vector to represent a 2D (3D) point

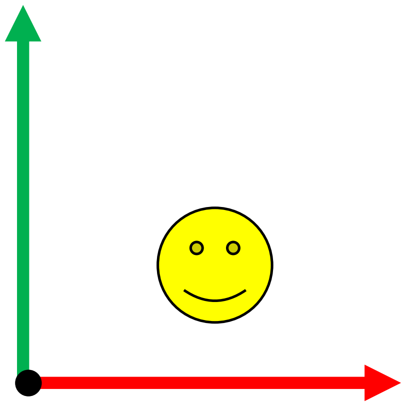
- Can concisely represent linear transformation & translation as matrix multiplication
    - Easier implementation

Combining affine transformations

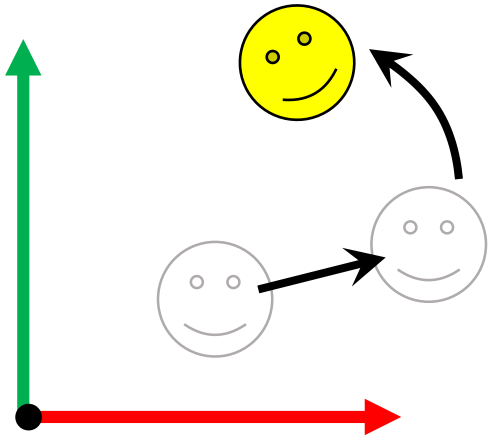
- Just multiply matrices
- Careful with the ordering!

$$R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

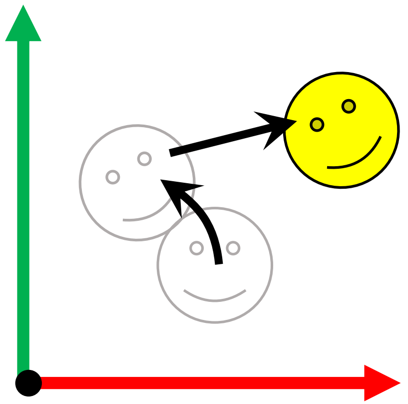
$$T = \begin{bmatrix} 1 & 0 & t_{\mathrm{x}} \\ 0 & 1 & t_{\mathrm{y}} \\ 0 & 0 & 1 \end{bmatrix}$$



$$\mathbf{x}' = R\,T\,\mathbf{x}$$



$$\mathbf{x}' = T\,R\,\mathbf{x}$$



Homogeneous coordinates

- When w≠0, 4D homogeneous coordinate (x, y, z, w) represents a 3D position $\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$

- Can represent **projective space** := 3D Euclid space + infinity points
  - When w→0, the represented 3D point approaches to infinity
    ➔ $(x, y, z, 0)$ represents a **directional vector** pointing toward (x, y, z)

  - Difference of positional vectors is a directional vector:
    $$(x, y, z, 1) - (x', y', z', 1) = (x - x', y - y', z - z', 0)$$

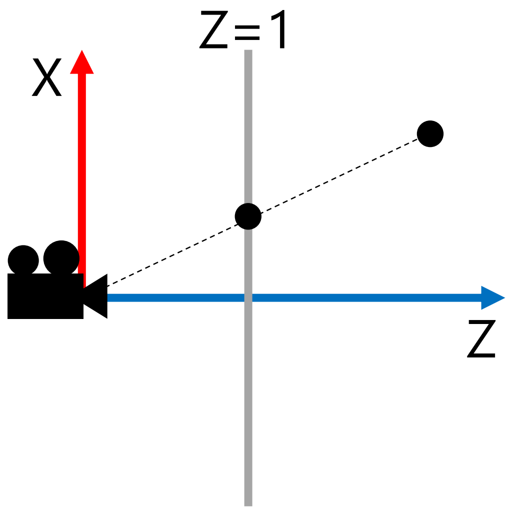
  - Homogeneous coordinate $(0, 0, 0, 0)$ is undefined

- More explanations in Wikipedia

Another role of homogeneous coordinates: Perspective projection

- An object’s apparent size on the screen is inverse proportional to the object-camera distance

- Camera at the origin, screen on the plane Z=1
  ➔ (p_x, p_y, p_z) is projected to $(w_x, w_y) = \left(\frac{p_x}{p_z}, \frac{p_x}{p_z}\right)$



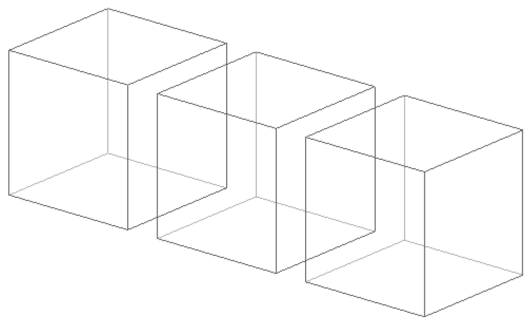
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z + 1 \\ p_z \end{bmatrix} \equiv \begin{bmatrix} p_x/p_z \\ p_y/p_z \\ 1 + 1/p_z \\ 1 \end{bmatrix} \begin{matrix} \to w_x \\ \to w_y \\ \to w_z \\ \end{matrix}$$

Projection matrix

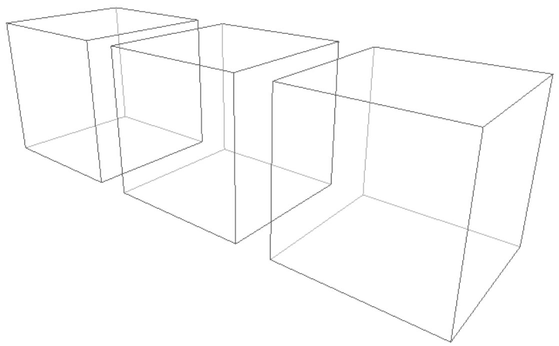
- w_z (depth value) is used for occlusion test ➔ Z-buffering

Orthographic projection

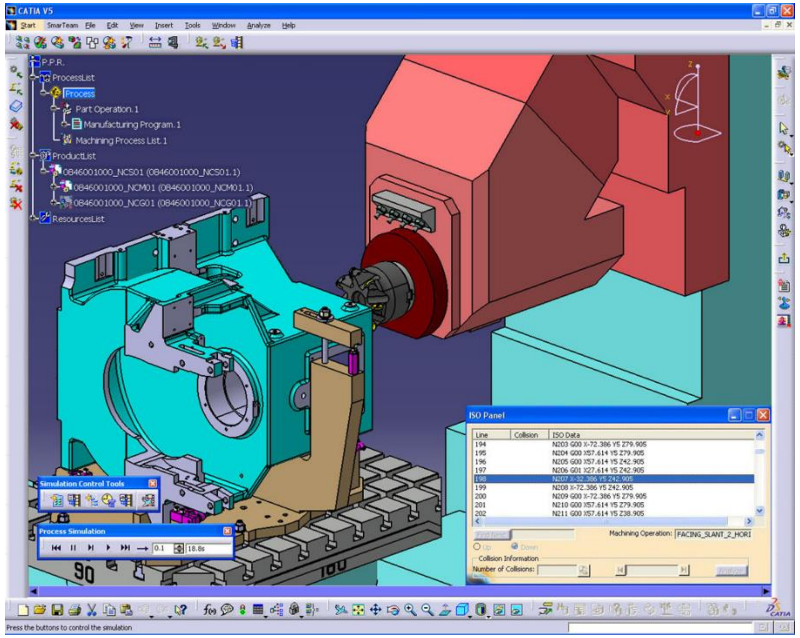
- Objects' apparent sizes don't depend on the camera position
- Simply ignore Z coordinates
- Frequently used in CAD



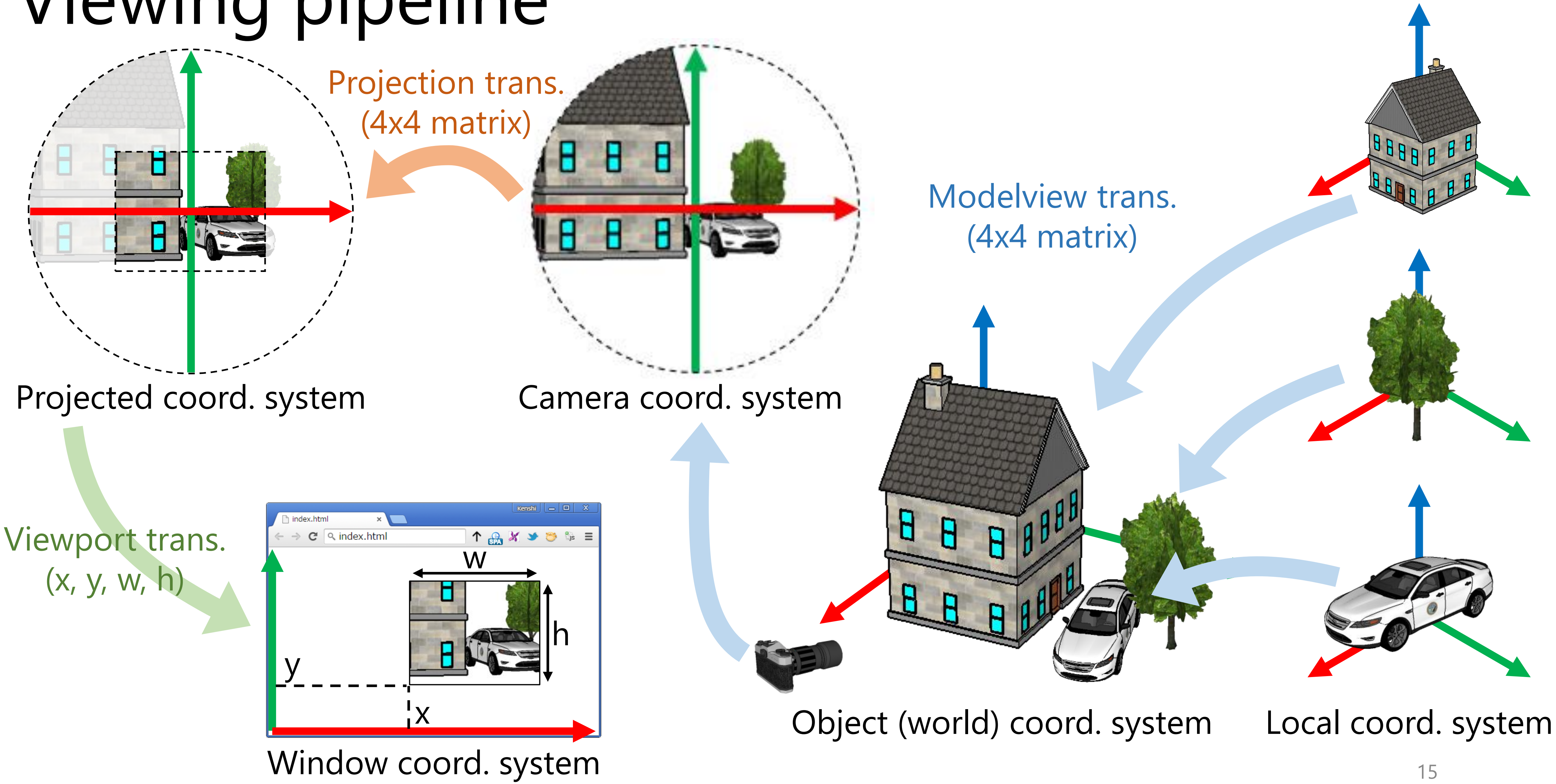
Orthographic



Perspective



Viewing pipeline



Classical OpenGL code

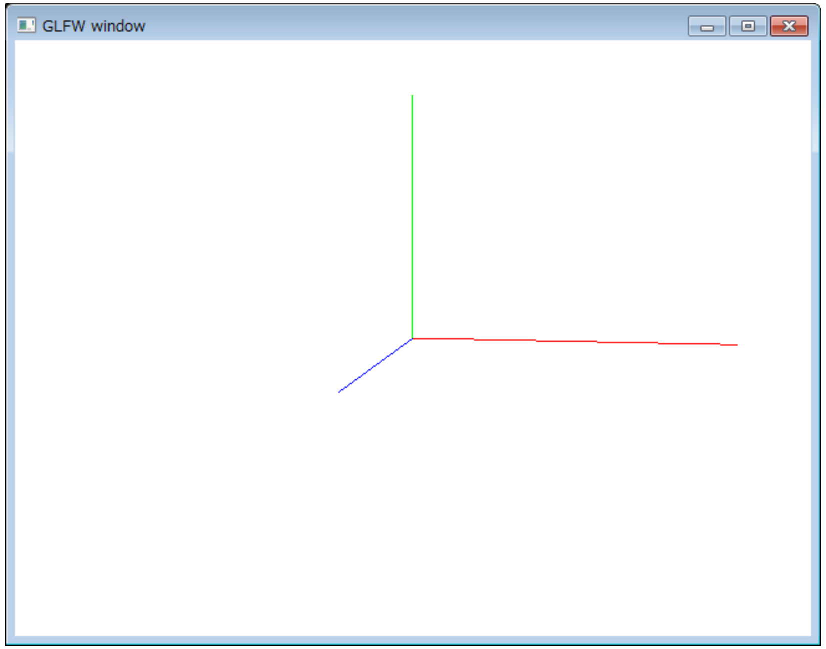
```
glViewport(0, 0, 640, 480);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(
  45.0,         // field of view
  640 / 480,    // aspect ratio
  0.1, 100.0); // depth range
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(
  0.5, 0.5, 3.0,   // view point
  0.0, 0.0, 0.0,   // focus point
  0.0, 1.0, 0.0); // up vector
glBegin(GL_LINES);
glColor3d(1, 0, 0); glVertex3d(0, 0, 0); glVertex3d(1, 0, 0);
glColor3d(0, 1, 0); glVertex3d(0, 0, 0); glVertex3d(0, 1, 0);
glColor3d(0, 0, 1); glVertex3d(0, 0, 0); glVertex3d(0, 0, 1);
glEnd();
```

Viewport transform

Projection transform

Modelview transform

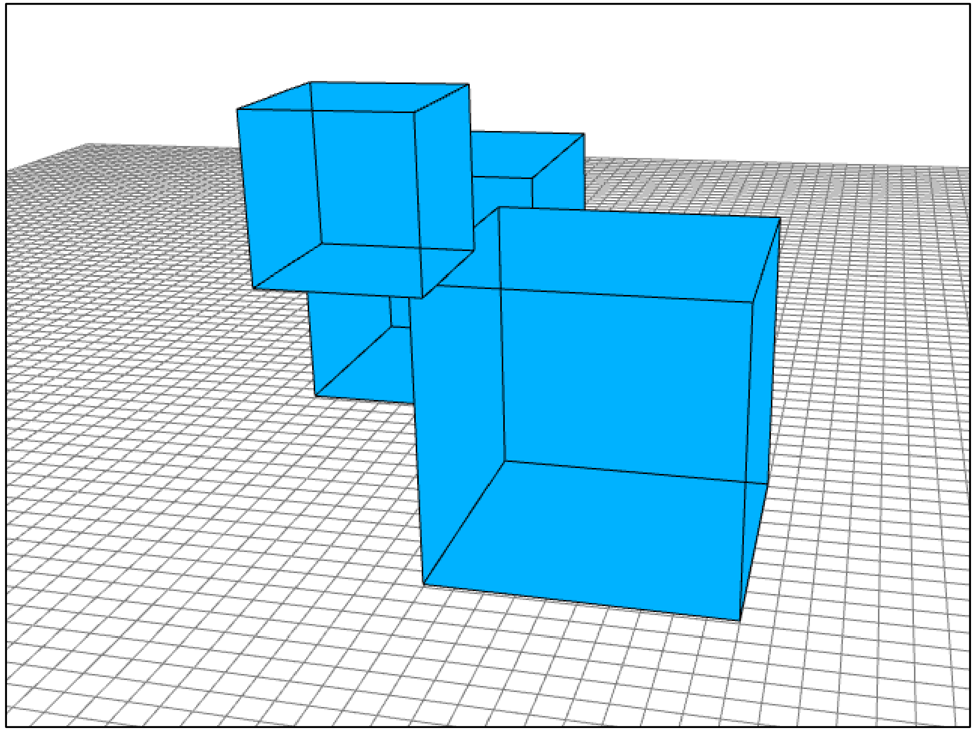
Scene content



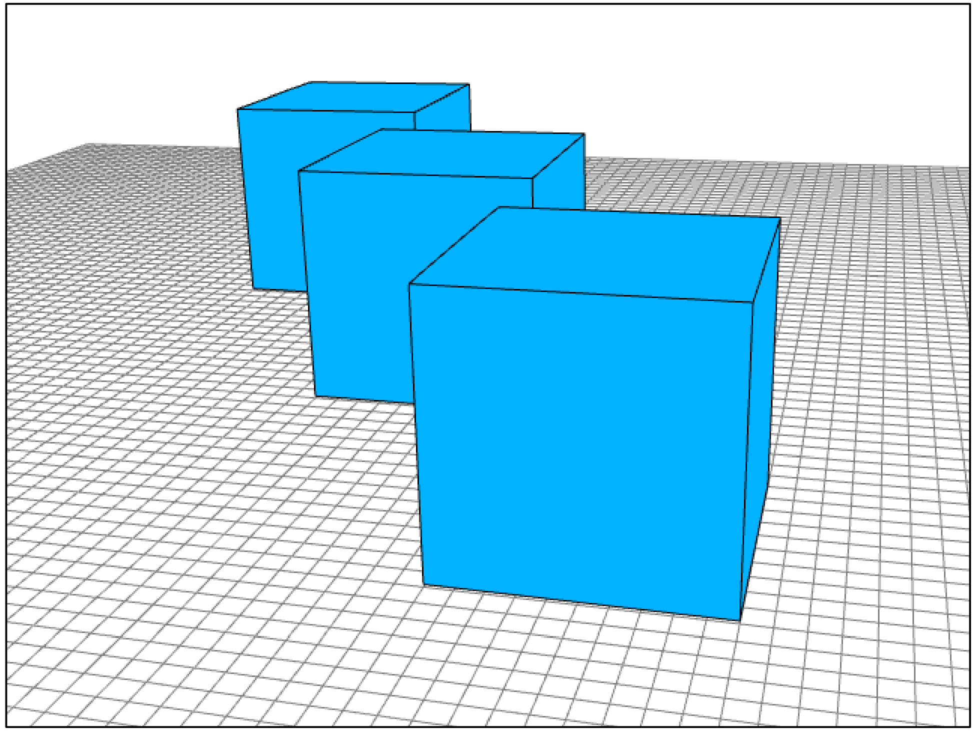
Output

Z-buffering

Hidden surface removal



Without hidden surface removal

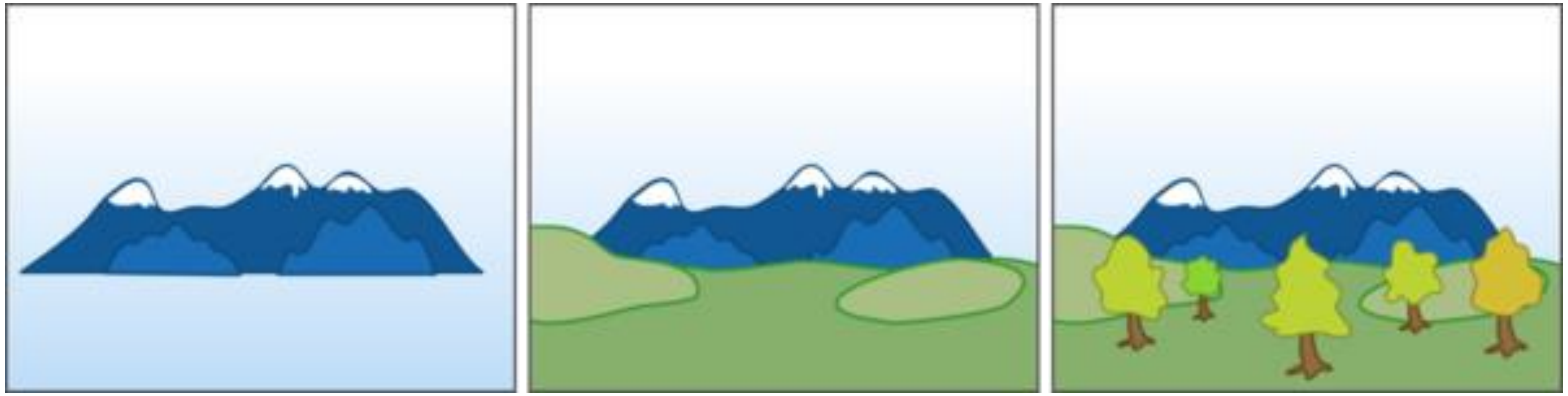


With hidden surface removal

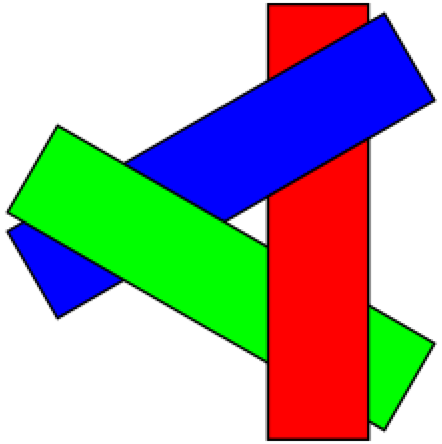
- Classic problem in CG

Painter's algorithm

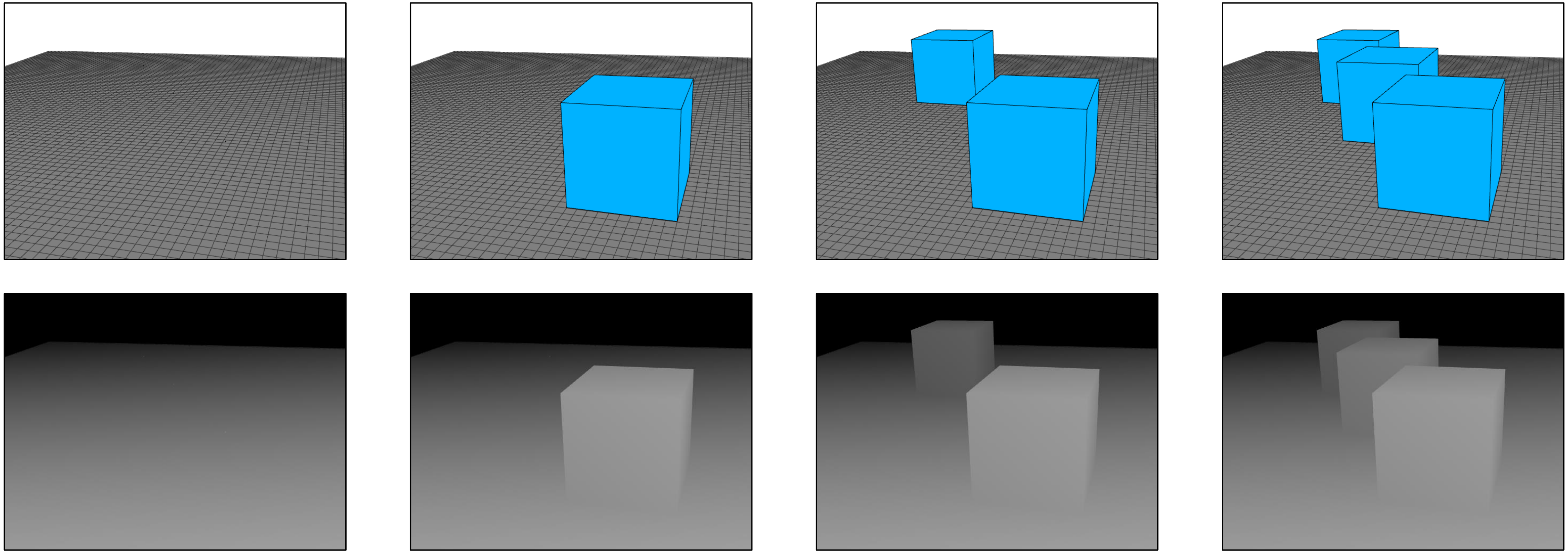
- Sort objects according to distances to camera, then draw them in the back-to-front order



- Fundamentally ill-suited for many cases
  - Sorting is also not always straightforward



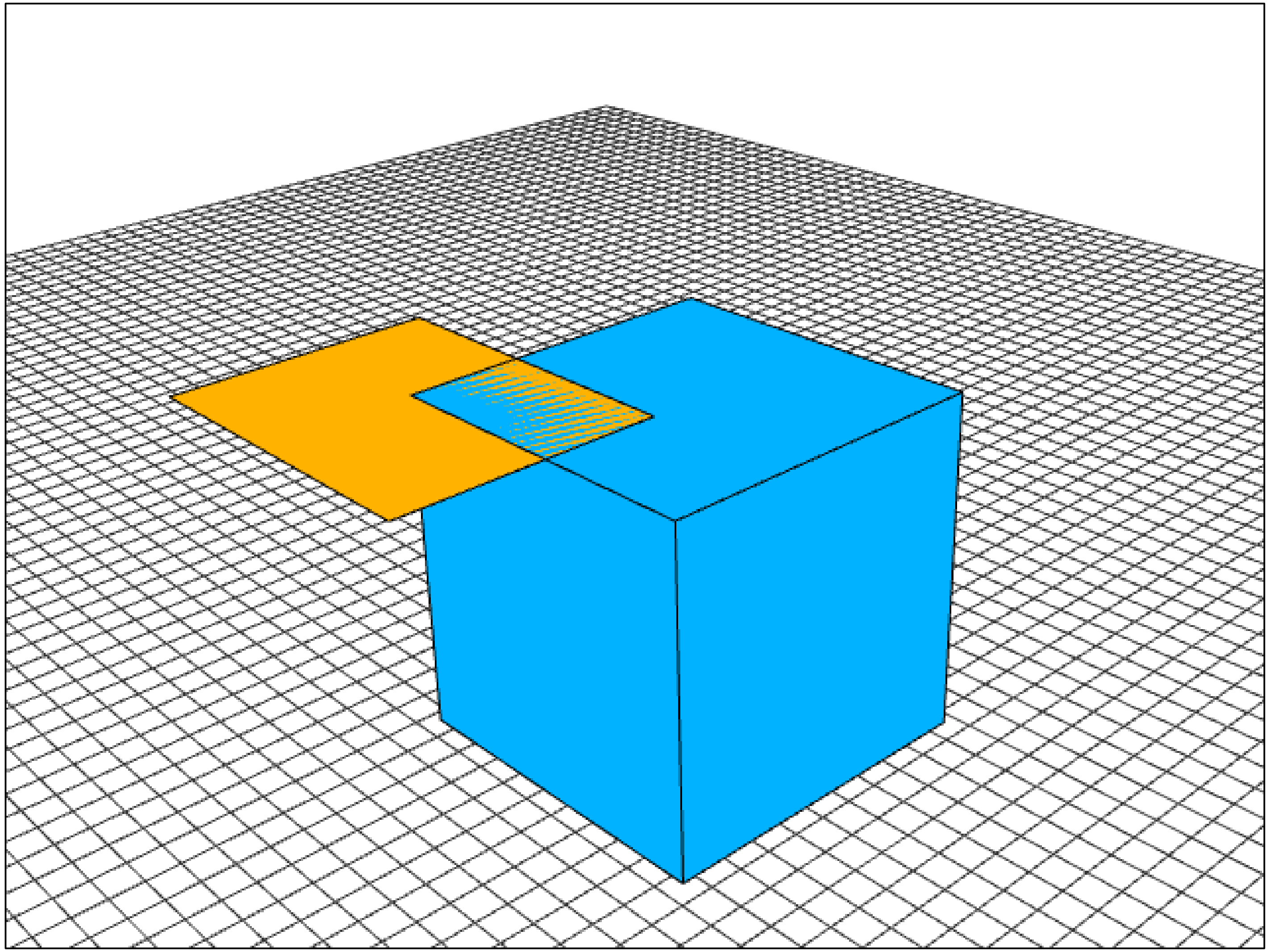
Z-buffering



- For each pixel, store distance to the camera (depth)
- More memory-consuming, but today's standard

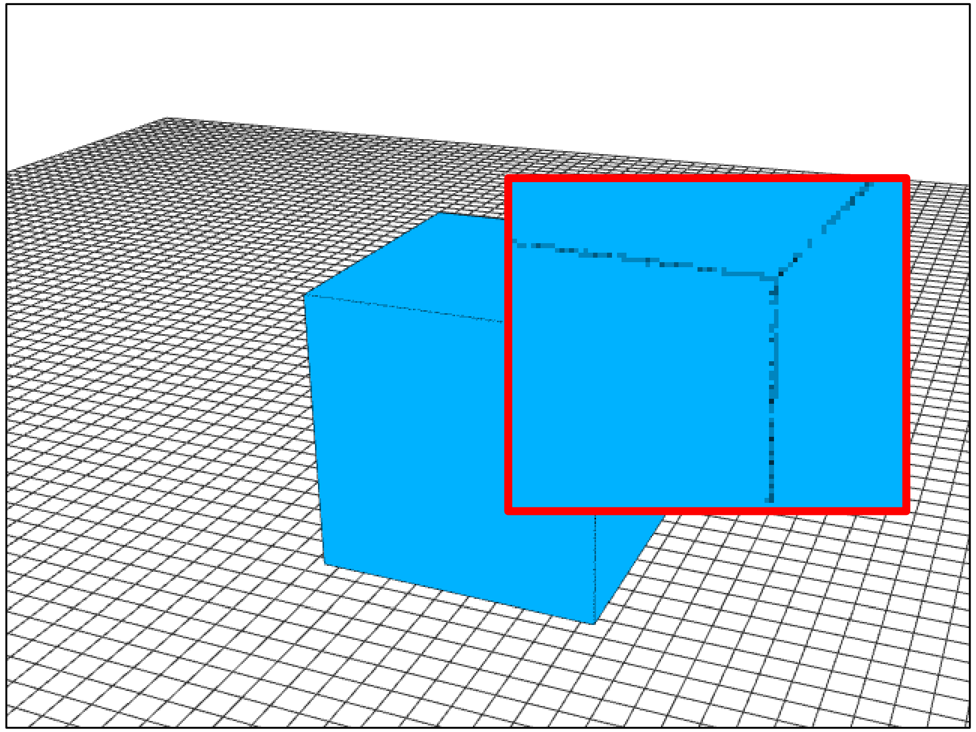
Typical issues with Z-buffering: Z-fighting

- Multiple polygons at exact same position
- Impossible to determine which is front/back
- Strange patterns due to rounding errors

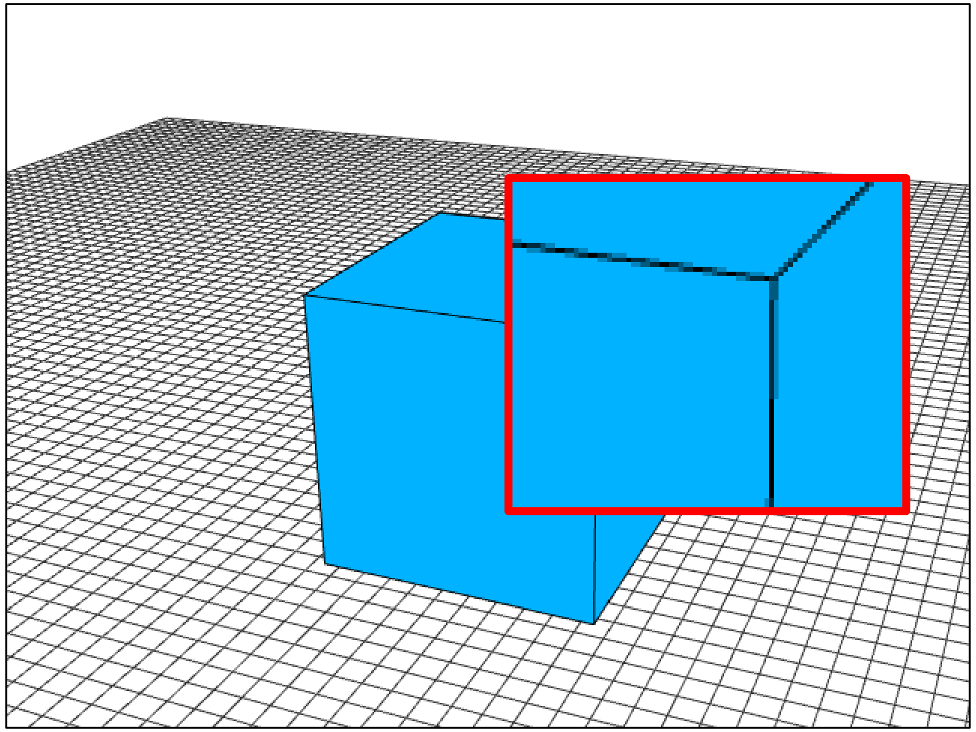


Typical issues with Z-buffering: Simultaneous drawing of faces and lines

- Dedicated OpenGL trick: glPolygonOffset



Without polygon offset

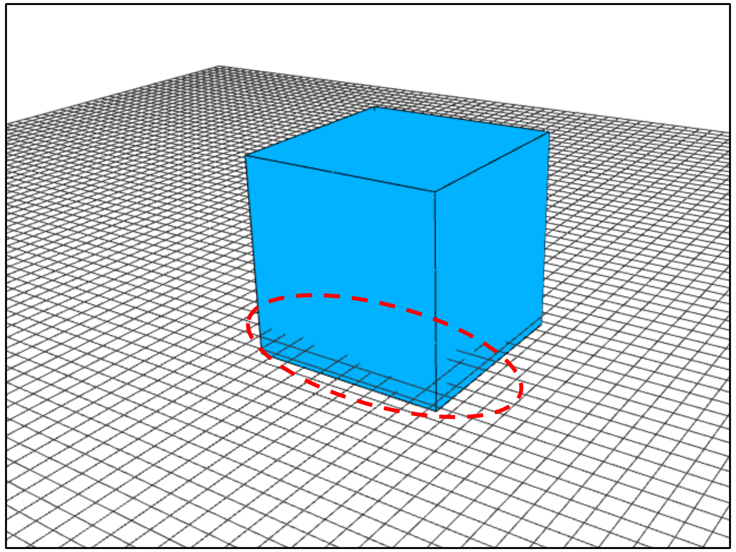


With polygon offset

Typical issues with Z-buffering: Depth range

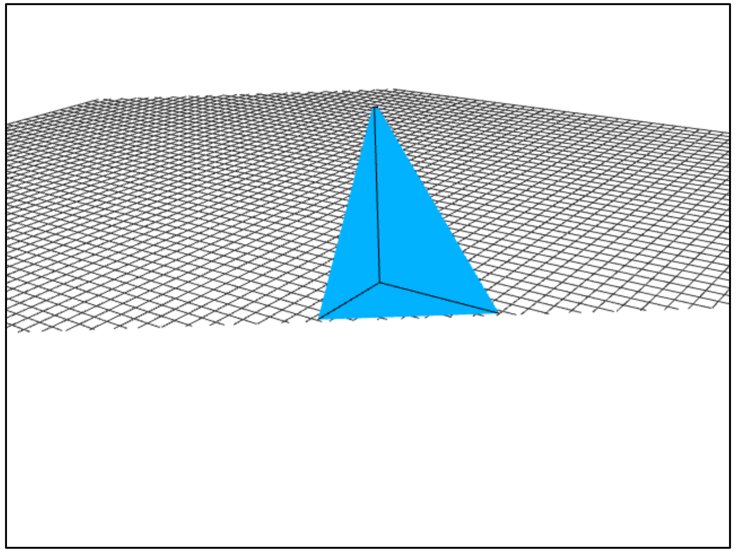
```
gluPerspective(
  45.0,           // field of view
  640 / 480,      // aspect ratio
  0.1, 1000.0); // zNear, zFar
```

- Fixed bits for Z-buffer
  - Typically, 16~24bits



zNear=0.0001
zFar =1000

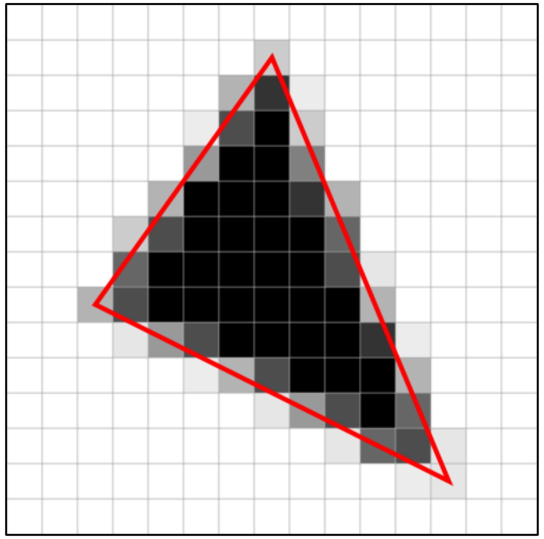
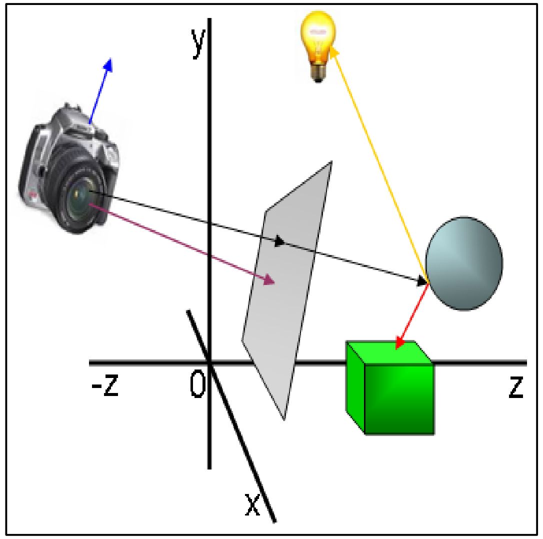
- Larger depth range
  ➔ Larger drawing space, less accuracy



zNear=50
zFar =100

- Smaller depth range
  ➔ More accuracy, smaller drawing space (clipped)

Rasterization vs Ray-tracing

	Rasterization	Ray-tracing
Purpose	Real-time CG (games)	High-quality CG (movies)
Idea	Per-polygon processing  One polygon updates multiple pixels	Per-pixel (ray) processing  One ray interacts with multiple polygons
Hidden surface removal	Z-buffering (OpenGL / DirectX)	By nature

More details by Prof. Hachisuka

Quaternions

Rotation about arbitrary axis

- Needed in various situations (e.g. camera manipulation)

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

about X-axis

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

about Y-axis

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

about Z-axis

$$R = \begin{bmatrix} \cos\theta + u_x^2\left(1-\cos\theta\right) & u_x u_y\left(1-\cos\theta\right) - u_z\sin\theta & u_x u_z\left(1-\cos\theta\right) + u_y\sin\theta \\ u_y u_x\left(1-\cos\theta\right) + u_z\sin\theta & \cos\theta + u_y^2\left(1-\cos\theta\right) & u_y u_z\left(1-\cos\theta\right) - u_x\sin\theta \\ u_z u_x\left(1-\cos\theta\right) - u_y\sin\theta & u_z u_y\left(1-\cos\theta\right) + u_x\sin\theta & \cos\theta + u_z^2\left(1-\cos\theta\right) \end{bmatrix}.$$

about arbitrary axis

- Matrix representation is overly complex!
  - Should be represented by 2 DoF (axis direction) + 1 DoF (angle) = 3 DoF

Degree **o**f **F**reedom

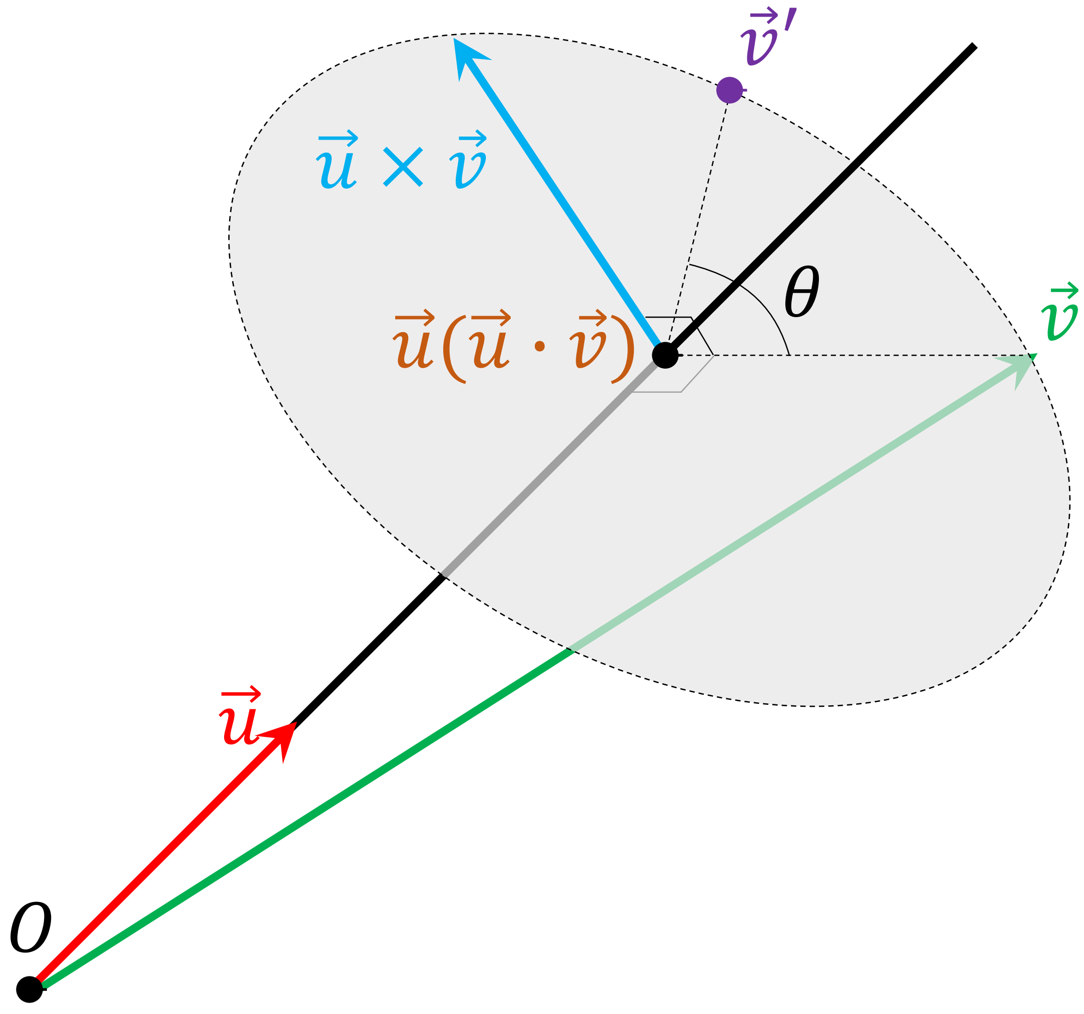
Geometry of axis-angle rotation

$\vec{u}$: axis (unit vector)

θ: angle

$\vec{v}$: input position

$\vec{v}'$: output position



$$\vec{v}' = \left(\vec{v} - \vec{u}(\vec{u} \cdot \vec{v})\right)\cos\theta + (\vec{u} \times \vec{v})\sin\theta + \vec{u}(\vec{u} \cdot \vec{v})$$

Complex number & quaternion

- Complex number
  - $\mathbf{i}^2 = -1$
  - $\mathbf{c} = (a, b) \coloneqq a + b\,\mathbf{i}$
  - $\mathbf{c}_1\mathbf{c}_2 = (a_1, b_1)(a_2, b_2) = a_1 a_2 - b_1 b_2 + (a_1 b_2 + b_1 a_2)\,\mathbf{i}$

- Quaternion
  - $\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$
    - $\mathbf{ij} = -\mathbf{ji} = \mathbf{k}, \quad \mathbf{jk} = -\mathbf{kj} = \mathbf{i}, \quad \mathbf{ki} = -\mathbf{ik} = \mathbf{j}$ Not commutative!
  - $\mathbf{q} = (a, b, c, d) \coloneqq a + b\,\mathbf{i} + c\,\mathbf{j} + d\,\mathbf{k}$
  - $\mathbf{q}_1\mathbf{q}_2 = (a_1, b_1, c_1, d_1)(a_2, b_2, c_2, d_2)$

$$= (a_1 a_2 - b_1 b_2 - c_1 c_2 - d_1 d_2) + (a_1 b_2 + b_1 a_2 + c_1 d_2 - d_1 c_2)\,\mathbf{i}$$

$$+ (a_1 c_2 + c_1 a_2 + d_1 b_2 - b_1 d_2)\,\mathbf{j} + (a_1 d_2 + d_1 a_2 + b_1 c_2 - c_1 b_2)\,\mathbf{k}$$

Notation by scalar + 3D vector

- $\mathbf{q} = a + b\,\mathbf{i} + c\,\mathbf{j} + d\,\mathbf{k} \coloneqq a + (b, c, d) = a + \vec{v}$
- $\mathbf{q}_1\mathbf{q}_2 = (a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2) + (a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2)\,\mathbf{i}$

$$+ (a_1c_2 + c_1a_2 + d_1b_2 - b_1d_2)\,\mathbf{j} + (a_1d_2 + d_1a_2 + b_1c_2 - c_1b_2)\,\mathbf{k}$$

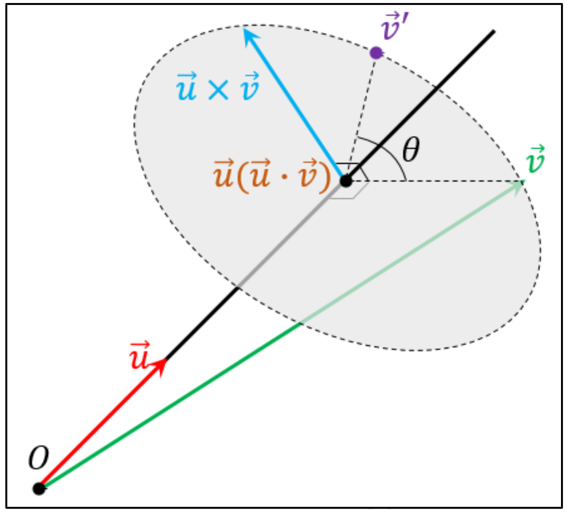
$$= (a_1a_2 - \overrightarrow{v_1} \cdot \overrightarrow{v_2}) + a_1\overrightarrow{v_2} + a_2\overrightarrow{v_1} + \overrightarrow{v_1} \times \overrightarrow{v_2}$$

Rotation using quaternions

$$q = \cos\frac{\alpha}{2} + \vec{u}\sin\frac{\alpha}{2}$$

Note: $\vec{u}$ is a unit vector

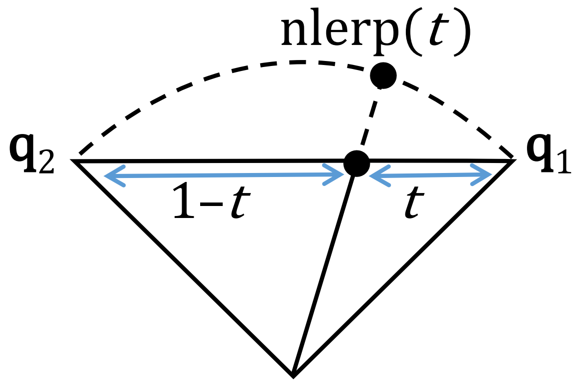
$$\begin{aligned}
\vec{v}' = q\vec{v}q^{-1} &= \left(\cos\frac{\alpha}{2} + \vec{u}\sin\frac{\alpha}{2}\right)\vec{v}\left(\cos\frac{\alpha}{2} - \vec{u}\sin\frac{\alpha}{2}\right) \\
&= \vec{v}\cos^2\tfrac{\alpha}{2} + (\vec{u}\vec{v} - \vec{v}\vec{u})\sin\tfrac{\alpha}{2}\cos\tfrac{\alpha}{2} - \vec{u}\vec{v}\vec{u}\sin^2\tfrac{\alpha}{2} \\
&= \vec{v}\cos^2\tfrac{\alpha}{2} + 2(\vec{u}\times\vec{v})\sin\tfrac{\alpha}{2}\cos\tfrac{\alpha}{2} - (\vec{v}(\vec{u}\cdot\vec{u}) - 2\vec{u}(\vec{u}\cdot\vec{v}))\sin^2\tfrac{\alpha}{2} \\
&= \vec{v}(\cos^2\tfrac{\alpha}{2} - \sin^2\tfrac{\alpha}{2}) + (\vec{u}\times\vec{v})(2\sin\tfrac{\alpha}{2}\cos\tfrac{\alpha}{2}) + \vec{u}(\vec{u}\cdot\vec{v})(2\sin^2\tfrac{\alpha}{2}) \\
&= \vec{v}\cos\alpha + (\vec{u}\times\vec{v})\sin\alpha + \vec{u}(\vec{u}\cdot\vec{v})(1-\cos\alpha) \\
&= \left(\vec{v} - \vec{u}(\vec{u}\cdot\vec{v})\right)\cos\alpha + (\vec{u}\times\vec{v})\sin\alpha + \vec{u}(\vec{u}\cdot\vec{v})
\end{aligned}$$



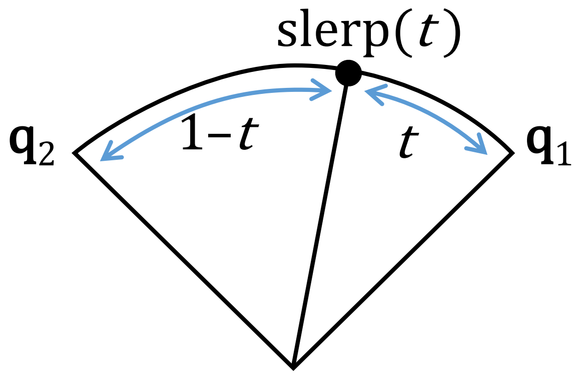
- Interesting theory behind (cf. Wikipedia)

Rotation interpolation using quaternions

- Linear interp + normalization (nlerp)
  - $\mathrm{nlerp}(\mathbf{q}_1, \mathbf{q}_2, t) \coloneqq \mathrm{normalize}\big((1-t)\mathbf{q}_1 + t\,\mathbf{q}_2\big)$
  - ☺less computation, ☹non-uniform angular speed



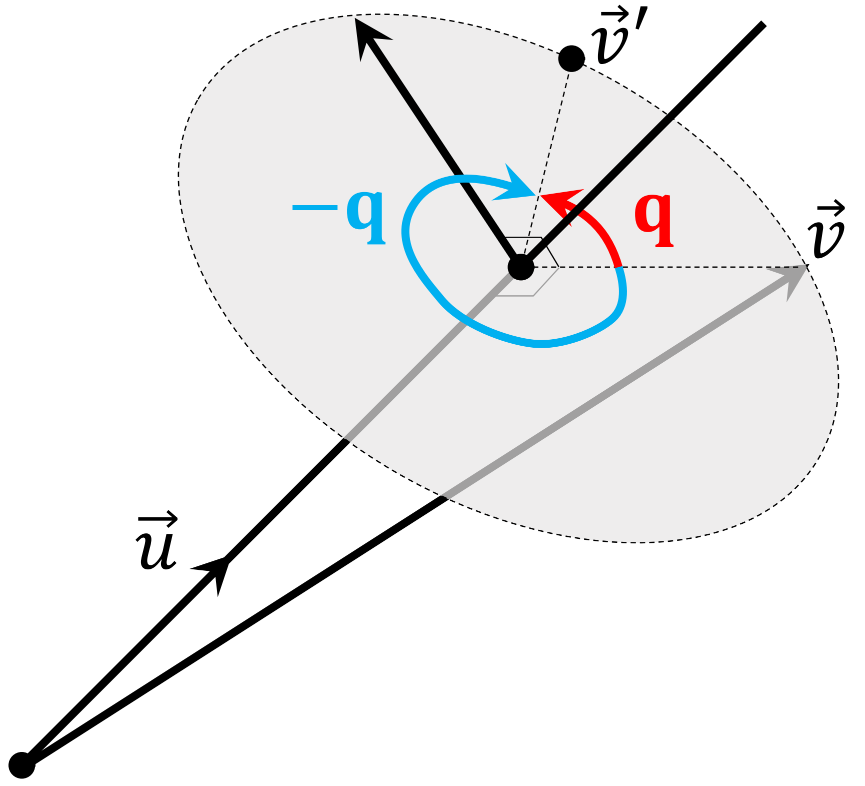
- Spherical linear interpolation (slerp)
  - $\Omega = \cos^{-1}(\mathbf{q}_1 \cdot \mathbf{q}_2)$
  - $\mathrm{slerp}(\mathbf{q}_1, \mathbf{q}_2, t) \coloneqq \frac{\sin(1-t)\Omega}{\sin\Omega}\mathbf{q}_1 + \frac{\sin t\Omega}{\sin\Omega}\mathbf{q}_2$
  - ☹more computation, ☺constant angular speed



Animating rotation with quaternion curves. Shoemake, SIGGRAPH 1985
http://number-none.com/product/Understanding%20Slerp,%20Then%20Not%20Using%20It/

Signs of quaternions

- Quaternion with angle θ:
  - $\mathbf{q} = \cos\frac{\theta}{2} + \vec{u}\sin\frac{\theta}{2}$
- Quaternion with angle $\theta - 2\pi$:
  - $\cos\frac{\theta-2\pi}{2} + \vec{u}\sin\frac{\theta-2\pi}{2} = -\mathbf{q}$



- When interpolating from $\mathbf{q}_1$ to $\mathbf{q}_2$, negate $\mathbf{q}_2$ if $\mathbf{q}_1 \cdot \mathbf{q}_2$ is negative
  - Otherwise, the interpolation path becomes longer

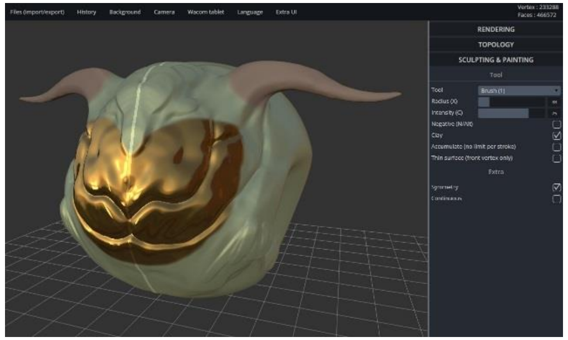
How to work on assignments

Choices for implementing real-time CG

- Two kinds of APIs for using GPU
  - Different API designs (slightly?)
  - Both supported by most popular programming languages
- Many choices for system- & langualge-dependent parts
  - GUI management, handling images, …
  - Many libraries:
    - GUI: GLUT (C), GLFW (C), SDL (C), Qt (C++), MFC (C++), wxWidgets (C++), Swing (Java), …
    - Images: libpng, OpenCV, ImageMagick
- Often quite some work to get started

WebGL™ = JavaScript + OpenGL®

- Runs on many (mobile) browsers
- HTML-based ➔ can easily handle multimedia & GUI
- No compiling!
  - Quick trial & error
- Some performance concerns
- Increasingly popular today



Hurdle in WebGL development: OpenGL ES

- No support for legacy OpenGL API
- Reasons:
  - Less efficient
  - Burden on hardware vendors

Immediate mode	glBegin, glVertex, glColor, glTexCoord
Polygonal primitives	GL_QUADS, GL_POLYGON
Light & material	glLight, glMaterial
Transform. matrices	GL_MODELVIEW, GL_PROJECTION
Display list	glNewList
Default shaders	

- Allowed API:

  Prepare arrays, send them to GPU, draw them using custom shaders

Shaders	glCreateShader, glShaderSource, glCompileShader, glCreateProgram, glAttachShader, glLinkProgram, glUseProgram
Shader variables	glGetAttribLocation, glEnableVertexAttribArray, glGetUniformLocation, glUniform
Arrays	glCreateBuffer, glBindBuffer, glBufferData, glVertexAttribPointer
Drawing	glDrawArrays

C / OpenGL 1.x

```c
#include <GL/glut.h>
void disp( void ) {
  float f;
  glClear(GL_COLOR_BUFFER_BIT);
  glPushMatrix();
  for(f = 0 ; f < 1 ; f += 0.1) {
    glColor3f(f , 0 , 0);
    glCallList(1);
  }
  glPopMatrix();
  glFlush();
}
void setDispList( void ) {
  glNewList(1, GL_COMPILE);
  glBegin(GL_POLYGON);
  glVertex2f(-1.2 , -0.9);
  glVertex2f(0.6 , -0.9);
  glVertex2f(-0.3 , 0.9);
  glEnd();
  glTranslatef(0.1 , 0 , 0);
  glEndList();
}
int main(int argc , char ** argv) {
  glutInit(&argc , argv);
  glutInitWindowSize(400 , 300);
  glutInitDisplayMode(GLUT_RGBA);
  glutCreateWindow("Kitty on your lap");
  glutDisplayFunc(disp);
  setDispList();
  glutMainLoop();
}
```

http://wisdom.sakura.ne.jp/system/opengl/gl20.html

WebGL

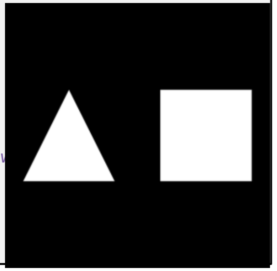
```html
<html><head>
<title>Learning WebGL &mdash; lesson 1</title>
<script type="text/javascript" src="glMatrix-0.9.5.mi
<script id="shader-fs" type="x-shader/x-fragment">
precision mediump float;
void main(void) {
  gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
</script>
<script id="shader-vs" type="x-shader/x-vertex">
attribute vec3 aVertexPosition;
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
void main(void) {
  gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
}
</script>
<script type="text/javascript">
var gl;
function initGL(canvas) {
  gl = canvas.getContext
  gl.viewportWidth = can
  gl.viewportHeight = ca
}
function getShader(gl, i
  var shaderScript = doc
var str = "";
  var k = shaderScript.f
  while (k) {
    if (k.nodeType == 3)
      str += k.textConte
    }
    k = k.nextSibling;
  }
  var shader;
  if (shaderScript.type
    shader = gl.createSh
  } else if (shaderScrip
    shader = gl.createSh
  }
  gl.shaderSource(shader
  gl.compileShader(shade
  return shader;
}
var shaderProgram;
function initShaders() {
  var fragmentShader = getShader(gl, "shader-fs");
  var vertexShader = getShader(gl, "shader-vs");
  shaderProgram = gl.createProgram();
  gl.attachShader(shaderProgram, vertexShader);
  gl.attachShader(shaderProgram, fragmentShader);
  gl.linkProgram(shaderProgram);
  gl.useProgram(shaderProgram);
  shaderProgram.vertexPositionAttribute =
    gl.getAttribLocation(shaderProgram, "aVertexPosition");
  gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
  shaderProgram.pMatrixUniform =
    gl.getUniformLocation(shaderProgram, "uPMatrix");
  shaderProgram.mvMatrixUniform =
    gl.getUniformLocation(shaderProgram, "uMVMatrix");
}
var mvMatrix = mat4.create();
var pMatrix = mat4.create();
```

```javascript
  0.0, 1.0, 0.0,
  -1.0, -1.0, 0.0,
  1.0, -1.0, 0.0
];
gl.bufferData(gl.ARRAY_BUFFER,
  new Float32Array(vertices),
  gl.STATIC_DRAW);
triangleVertexPositionBuffer.itemSize = 3;
triangleVertexPositionBuffer.numItems = 3;
squareVertexPositionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, squareVertexPositionBuffer);
vertices = [
  1.0, 1.0, 0.0,
  -1.0, 1.0, 0.0,
  1.0, -1.0, 0.0,
  -1.0, -1.0, 0.0
];
gl.bufferData(gl.ARRAY_BUFFER,
  new Float32Array(vertices),
  gl.STATIC_DRAW);
squareVertexPositionBuffer.itemSize = 3;
squareVertexPositionBuffer.numItems = 4;

drawScene() {
port(0, 0, gl.viewportWidth, gl.viewportHeight);
(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
spective(45, 1.5, 0.1, 100.0, pMatrix);
entity(mvMatrix);
anslate(mvMatrix, [-1.5, 0.0, -7.0]);
Buffer(gl.ARRAY_BUFFER, triangleVertexPositionBuffer);
xAttribPointer(shaderProgram.vertexPositionAttribute,
gleVertexPositionBuffer.itemSize,
OAT, false, 0, 0);
ixUniforms();
Arrays(gl.TRIANGLES, 0, triangleVertexPositionBuffer.numItems);
anslate(mvMatrix, [3.0, 0.0, 0.0]);
Buffer(gl.ARRAY_BUFFER, squareVertexPositionBuffer);
xAttribPointer(shaderProgram.vertexPositionAttribute,
VertexPositionBuffer.itemSize,
OAT, false, 0, 0);
ixUniforms();
Arrays(gl.TRIANGLE_STRIP, 0, squareVertexPositionBuffer.numItems);

ebGLStart() {
as = document.getElementById("lesson01-canvas");
anvas);
ders();
initBuffers();
  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  gl.enable(gl.DEPTH_TEST);
  drawScene();
}
</script></head>
<body onload="webGLStart();">
<canvas id="lesson01-canvas" style="border: none;"
height="500">
</canvas>
</body> </html>
```

```javascript
mat4.translate(mvMatrix, [-1.5,
gl.bindBuffer(gl.ARRAY_BUFFER,
gl.vertexAttribPointer(shaderPr
  triangleVertexPositionBuffer.
  gl.FLOAT, false, 0, 0);
setMatrixUniforms();
gl.drawArrays(gl.TRIANGLES, 0,
mat4.translate(mvMatrix, [3.0,
gl.bindBuffer(gl.ARRAY_BUFFER,
gl.vertexAttribPointer(shaderPr
  squareVertexPositionBuffer.it
  gl.FLOAT, false, 0, 0);
setMatrixUniforms();
gl.drawArrays(gl.TRIANGLE_STRIP
```



http://learningwebgl.com/blog/?p=28

Libraries for easing WebGL development

- Many popular ones:
  - three.js, O3D, OSG.JS, …
- All APIs are high-level, quite different from legacy OpenGL API☹
- Good for casual users, but maybe not for CS students (?)

three.js

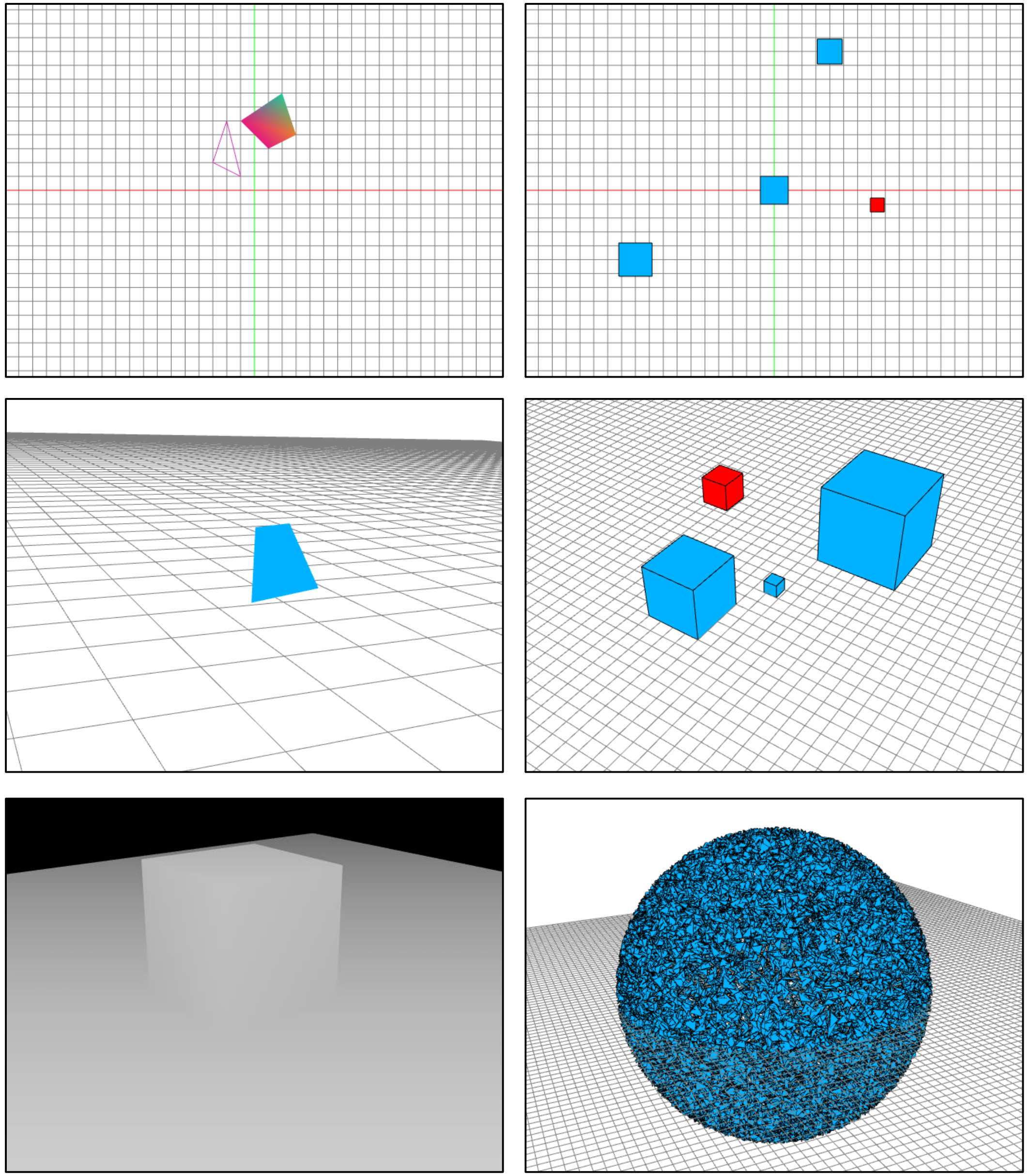
```
<script src="js/three.min.js"></script>
<script>
var camera, scene, renderer, geometry, material, mesh;
function init() {
  scene = new THREE.Scene();
  camera = new THREE.PerspectiveCamera( 75, 640 / 480, 1, 10000 );
  camera.position.z = 1000;
  geometry = new THREE.BoxGeometry( 200, 200, 200 );
  material = new THREE.MeshBasicMaterial({color:0xff0000, wireframe:true});
  mesh = new THREE.Mesh( geometry, material );
  scene.add( mesh );
  renderer = new THREE.WebGLRenderer();
  renderer.setSize(640, 480);
  document.body.appendChild( renderer.domElement );
}
function animate() {
  requestAnimationFrame( animate );
  render();
}
function render() {
  mesh.rotation.x += 0.01;
  mesh.rotation.y += 0.02;
  renderer.render( scene, camera );
}
init();
animate();
</script>
```

High-level API

legacygl.js

- Developed by me for this course
  - https://bitbucket.org/kenshi84/legacygl.js
  - Demos & tutorial

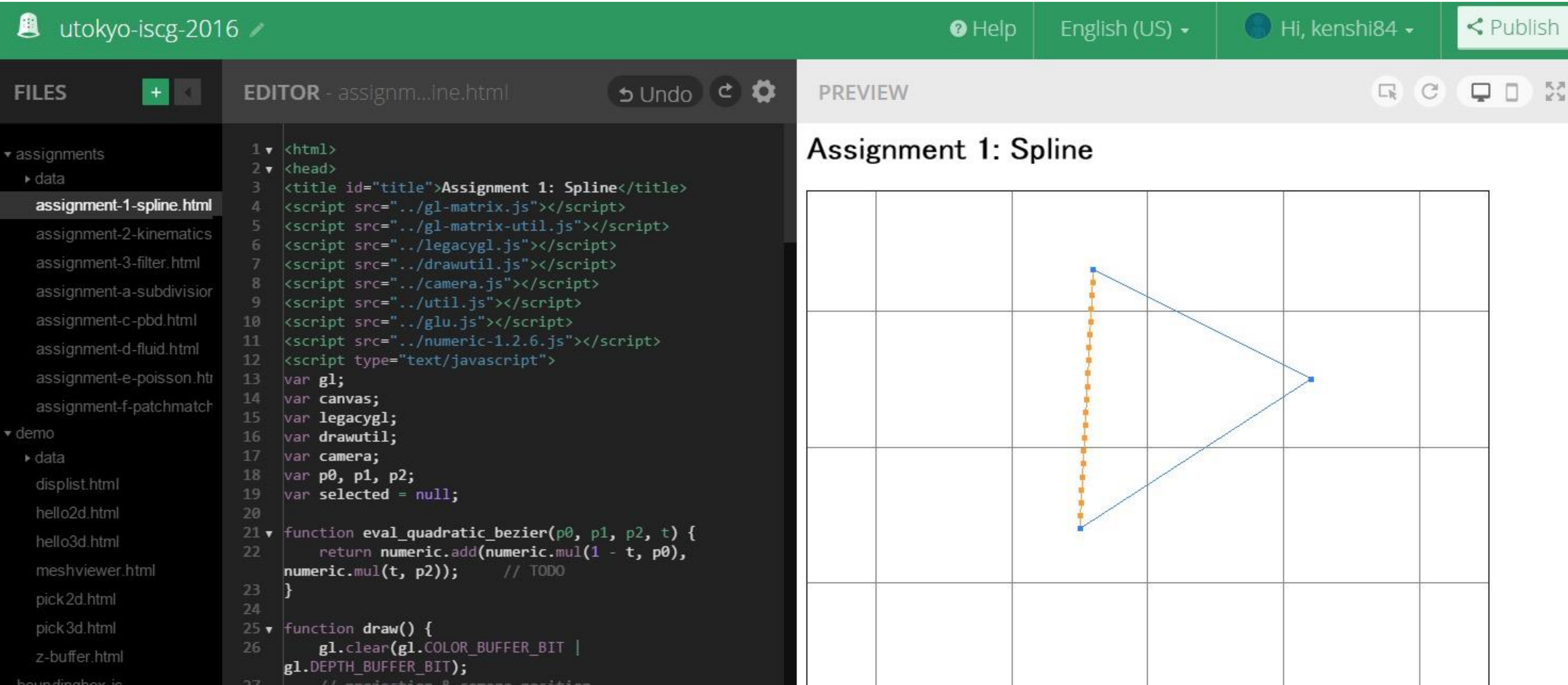
- Assignemnts' sample codes will be mostly using this
  - Try playing with it and see how it works



WebGL development using Mozilla Thimble

- A free web space for putting js/html/css
- Online editing and quick previewing

https://thimble.mozilla.org/



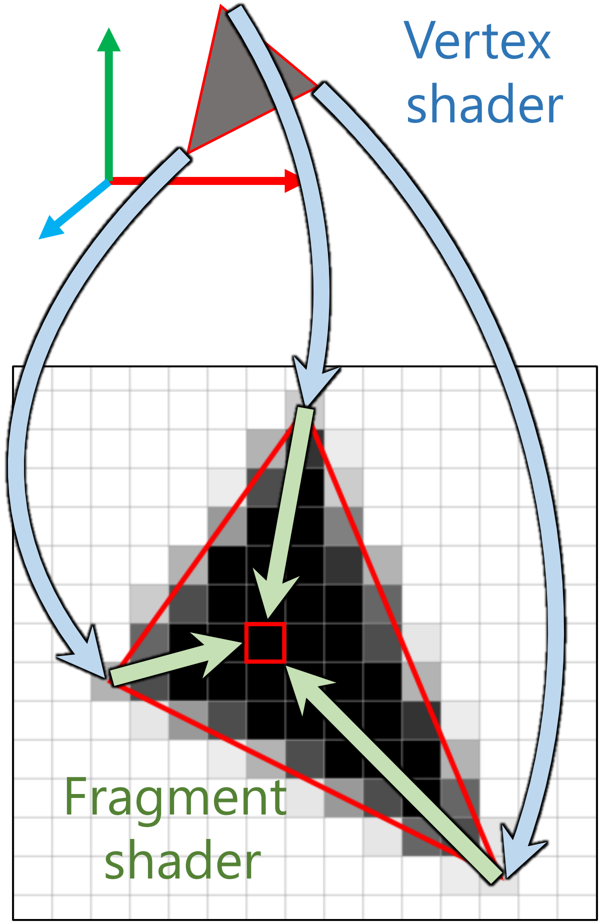
How to work on assignemnts

- Implement your solution using WebGL, upload it to the web, email the URL to the TA
  - Thimble is recommended, but other means (e.g. your own server) is also OK
  - Include some descriptions/discussions/etc in the HTML page
  - Other WebGL libraries than legacygl.js can also be used

- Other programming languages (e.g. C++) are also acceptable
  - Should compile and run on typical computing systems
  - Include source+binary+doc in a single ZIP file

- If anything unclear, contact TA or me

Shaders



- Vertex shader: per-vertex processing
  - Per-vertex data passed by glBufferData
    - Vertex position, color, texture coordinate, …
  - Mandatory operation: Specify vertex location on the screen after coordinate transformation (gl_Position)

- Fragment shader: per-pixel processing
  - Do something with rasterized (=linearly interpolated) data
  - Mandatory operation: Specify pixel color to be drawn (gl_FragColor)

- GLSL (Open**GL** **S**hading **L**anguage) codes passed to GPU as strings
  ➔ compiled at runtime

Shader variables

- uniform variables
  - Readable from vertex/fragment shaders
  - Passed to GPU separately from vertex arrays (glUniform)
  - Examples: modelview/projection matrices, flags
- attribute variables
  - Readable only from vertex shaders
  - Vertex array data passed to GPU via glBufferData
  - Examples: XYZ position, RGB color, UV texcoord
- varying variables
  - Written by vertex shader, read by fragment shader
  - Per-vertex data linearly interpolated at this pixel

(Grammer might change depending on versions)

```
uniform mat4 u_modelview;
uniform mat4 u_projection;
attribute vec3 a_vertex;
attribute vec3 a_color;
varying vec3 v_color;
void main(void) {
  gl_Position = u_projection
              * u_modelview
              * vec4(a_vertex, 1.0);
  v_color = a_color;
}
```

Vertex shader

```
precision mediump float;
varying vec3 v_color;
void main(void) {
  gl_FragColor.rgb = v_color;
  gl_FragColor.a   = 1.0;
}
```

Fragment shader

Tips for JavaScript beginners (=me)

- 7 types: String / Bool / Number / Function / Object / null / undefined
  - Unlike C++
- Number: always double precision
  - No distinction between integer & floating point
- Object: associative map with string keys
  - `x.abc` is equivalent to `x["abc"]` (as if a “member”)
  - `{ abc : y }` is equivalent to `{ "abc" : y }`
  - Non-string keys are implicitly converted to strings
- Arrays are special objects with keys being consecutive integers
  - With additional capabilities: `.length` , `.push()` , `.pop()` , `.forEach()`
- Always pass-by-value when assigning & passing arguments
  - No language support for “deep copy”
- When in doubt, use `console.log(x)`

References

- OpenGL
  - Official spec
    https://www.opengl.org/sdk/docs/man/html/indexflat.php
- WebGL/JavaScript/HTML5
  - Learning WebGL
    http://learningwebgl.com/blog/?p=11
  - Official spec
    https://www.khronos.org/registry/webgl/specs/1.0/#5.14
  - Mozilla Developer Network
    - https://developer.mozilla.org
  - An Introduction to JavaScript for Sophisticated Programmers
    http://casual-effects.blogspot.jp/2014/01/
  - Effective JavaScript
    http://effectivejs.com/

References

- http://en.wikipedia.org/wiki/Affine_transformation
- http://en.wikipedia.org/wiki/Homogeneous_coordinates
- http://en.wikipedia.org/wiki/Perspective_(graphical)
- http://en.wikipedia.org/wiki/Z-buffering
- http://en.wikipedia.org/wiki/Quaternion