Introduction to Computer Graphics

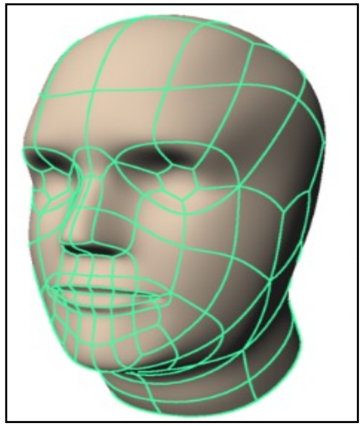
April 8, 2021

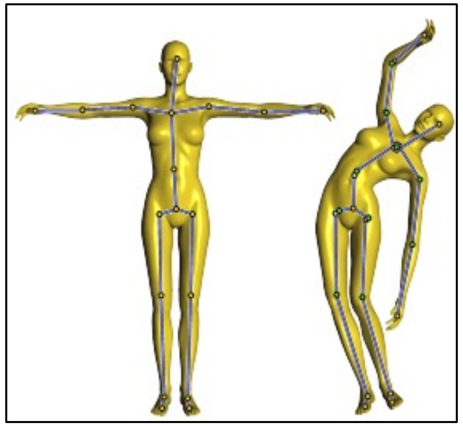
Kenshi Takayama

Course overview

- Lecture basic stuff on 4 topics
- 2~3 lectures per topic, 12 lectures in total

Modeling





Animation

Rendering

Image processing

Lecturer

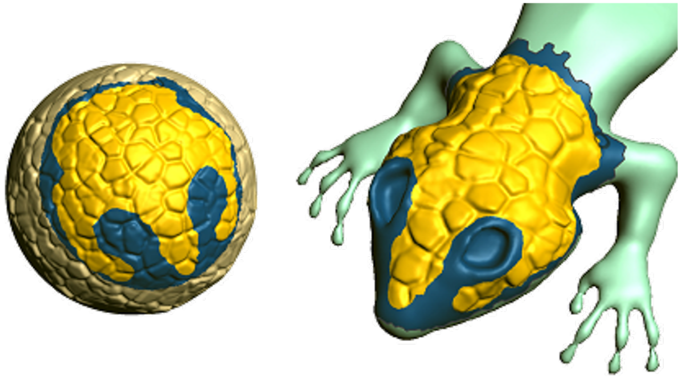
TA：Koki Endo @Igarashi Lab MSc
endo-k-88st0@g.ecc.u-tokyo.ac.jp

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

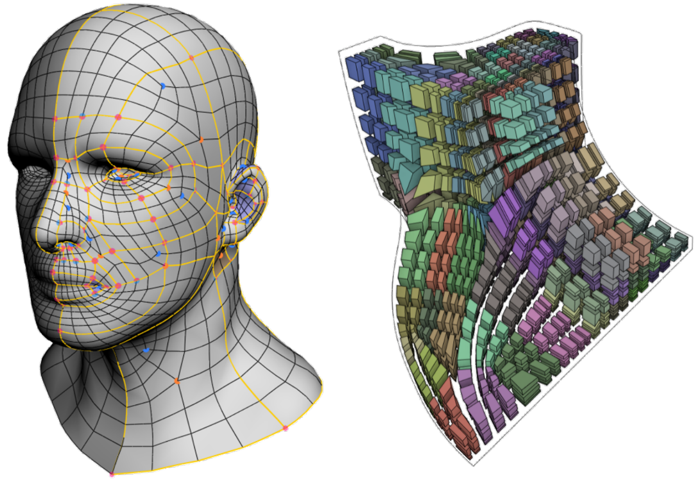
BSc+MSc+PhD 2003~2012 → PostDoc 2012~2014 → Assist Prof 2014~

ETH → NII

Modeling objects with internal structures



UI for 3D modeling



Generation of quad/hexahedral meshes

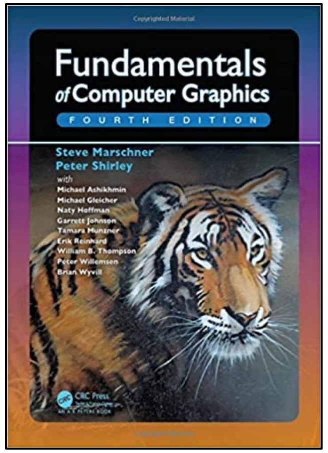
Grading

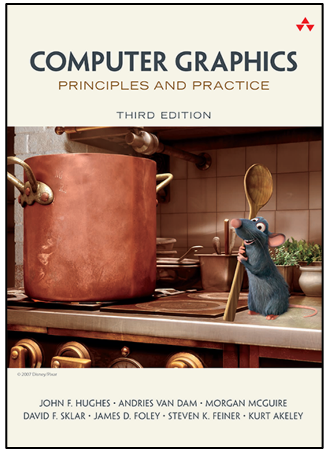
- Programming assignments only
  - No exam, no attendance check
- Two types of assignments: Basic & Advanced
  - Basic: 1 assignment per topic (4 in total), very easy
  - Advanced: For motivated students
- Deadline:
  - Basic: 2 weeks after announcement
  - Advanced: end of July
- Evaluation criteria
  - 1 assignment 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

Course information

- Website
  - http://research.nii.ac.jp/~takayama/teaching/utokyo-iscg-2021/

- Famous textbooks (not used in the class)
  - Fundamentals of Computer Graphics (978-1568814698)
  - Computer Graphics: Principles and Practice (978-0321399526)





Coordinate transformations

Linear transformation

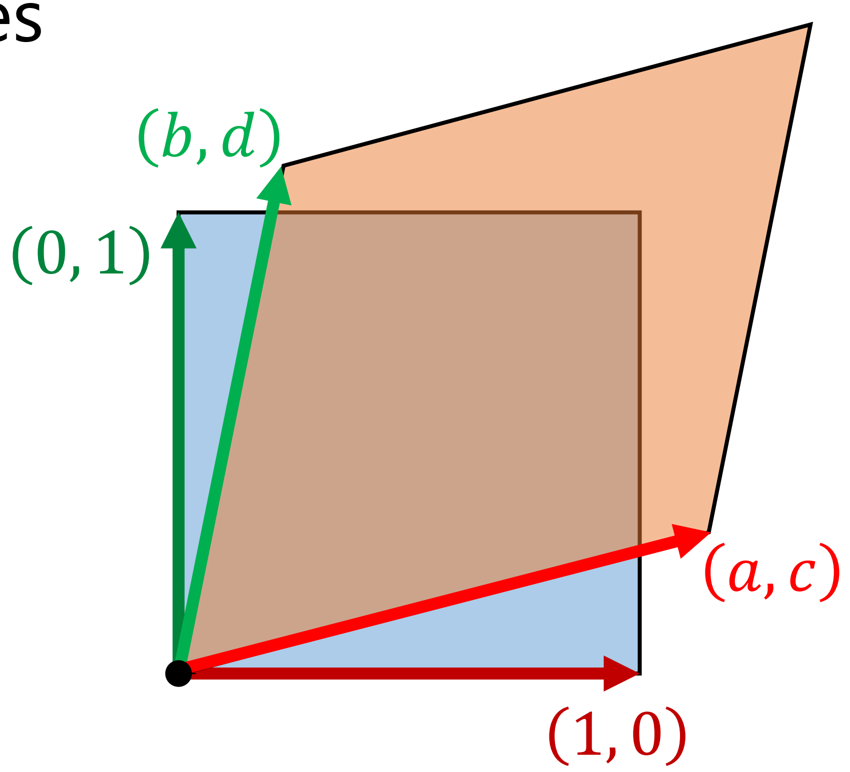
In 2D: $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$

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

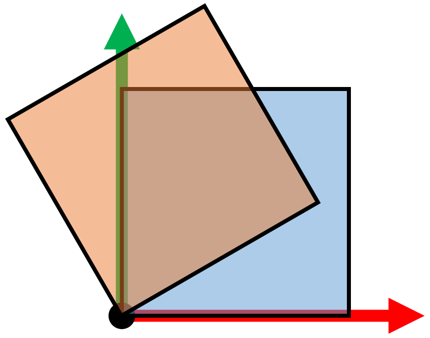
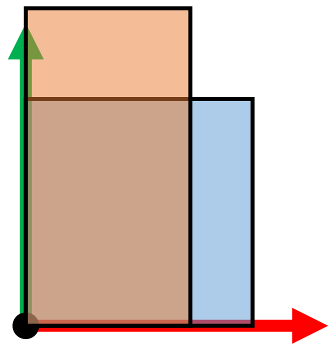
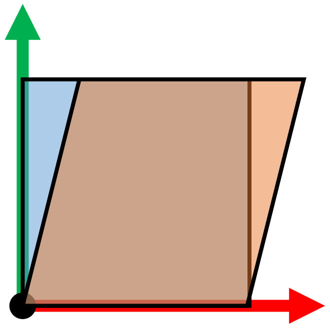
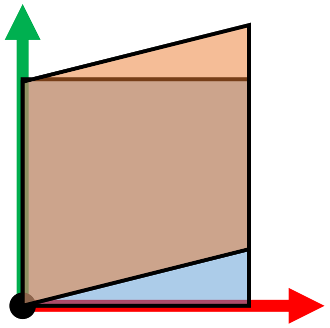
$$\begin{bmatrix} a \\ c \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

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



- Origin stays put

Special linear transformations

Rotation	Scaling	Shearing (X dir.)	Shearing (Y dir.)
			
$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$	$\begin{bmatrix} s_{\mathrm{x}} & 0 \\ 0 & s_{\mathrm{y}} \end{bmatrix}$	$\begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 \\ k & 1 \end{bmatrix}$

Linear transformation + translation = Affine transformation

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

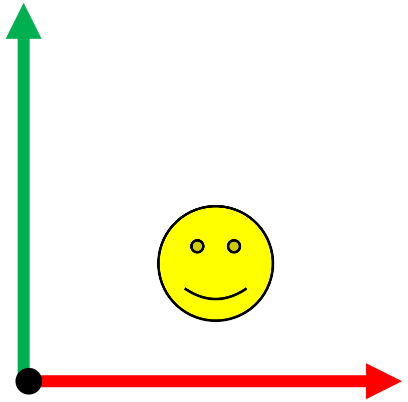
- Homogeneous coordinates: Use a 3D (4D) vector to represent a 2D (3D) point
- Can concisely represent linear transformation & translation as matrix multiplication
  - Easier implementation

Combining affine transformations

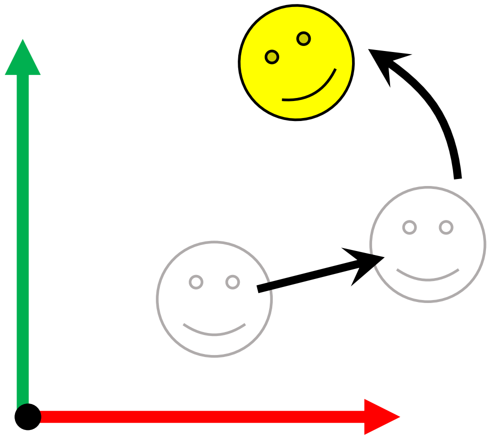
- Just multiply matrices
- Careful with the ordering!

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

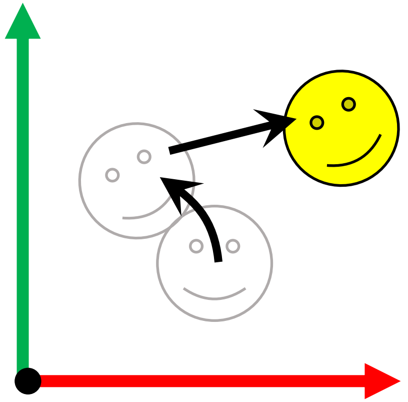
$$T = \begin{bmatrix} 1 & 0 & t_{\mathrm{x}} \\ 0 & 1 & t_{\mathrm{y}} \\ 0 & 0 & 1 \end{bmatrix}$$



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

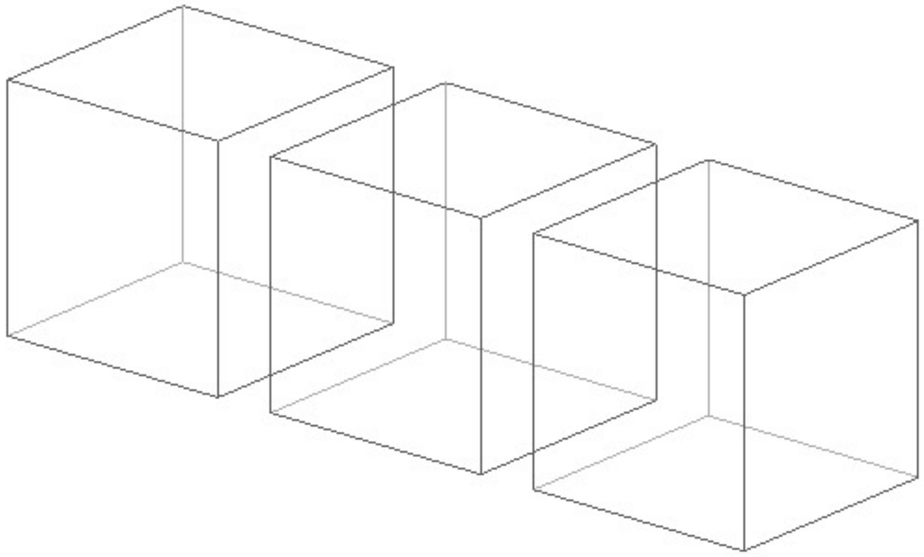


$$\mathbf{x}' = T\,R\,\mathbf{x}$$

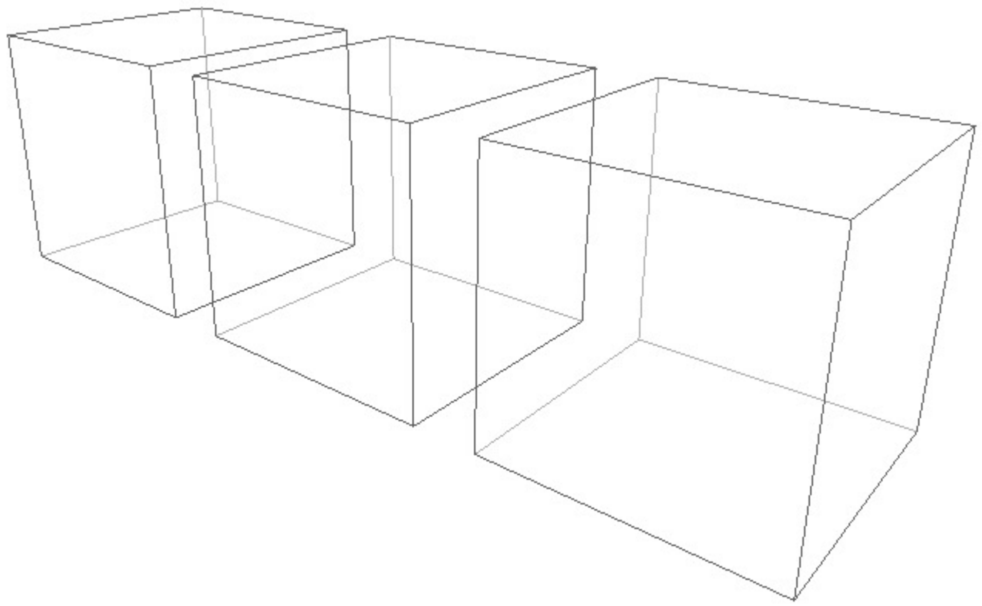


Another role of homogeneous coordinates: Perspective projection

- Objects' apparent sizes on the screen are inversely proportional to the object-camera distance



Orthographic projection



Perspective projection

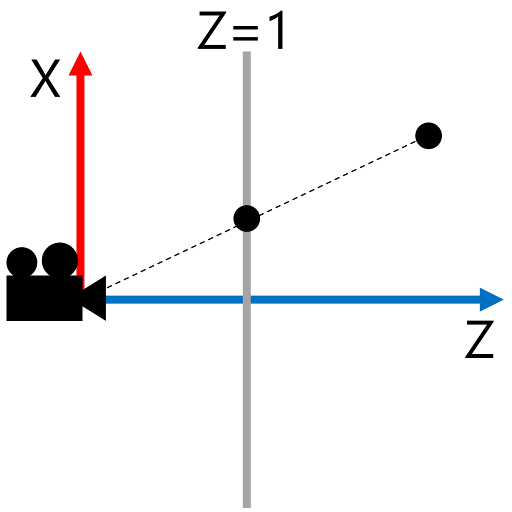
Another role of homogeneous coordinates: Perspective projection

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

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

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

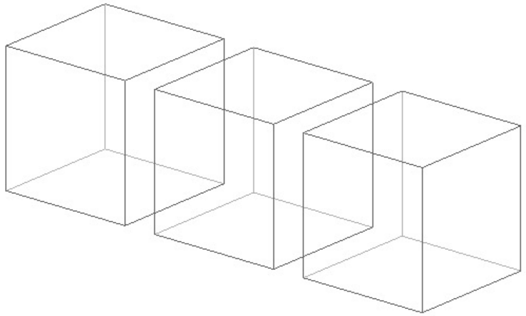
Projection matrix



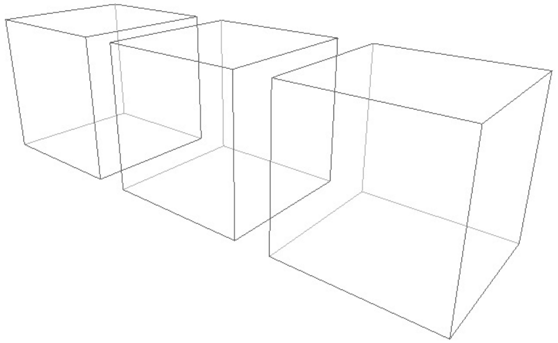
- w_z (depth value) is used for occlusion test ➔ Z-buffering

Orthographic projection

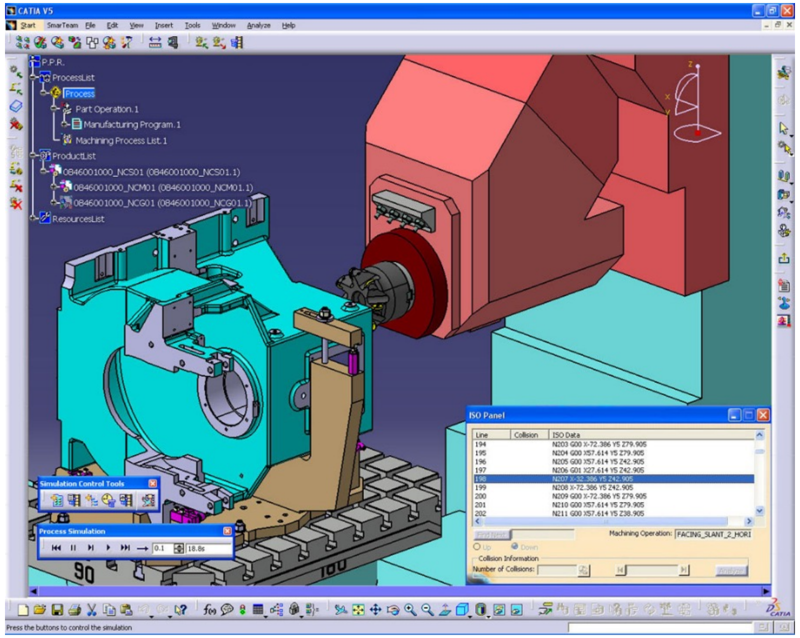
- Objects' apparent sizes don't depend on the camera position
- Simply ignore Z coordinates
- Frequently used in CAD



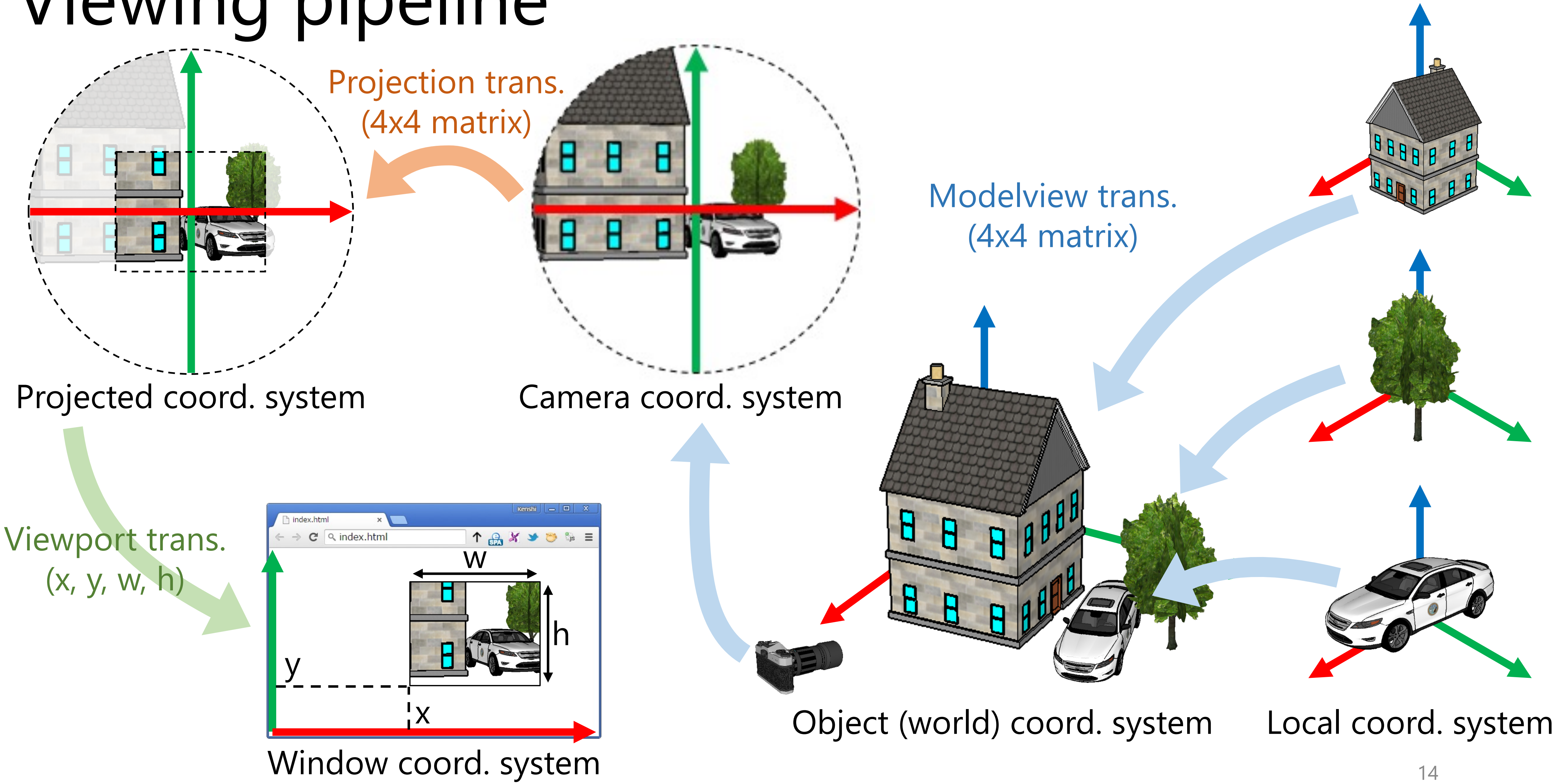
Orthographic



Perspective



Viewing pipeline



Classical OpenGL code

```
glViewport(0, 0, 640, 480);
```

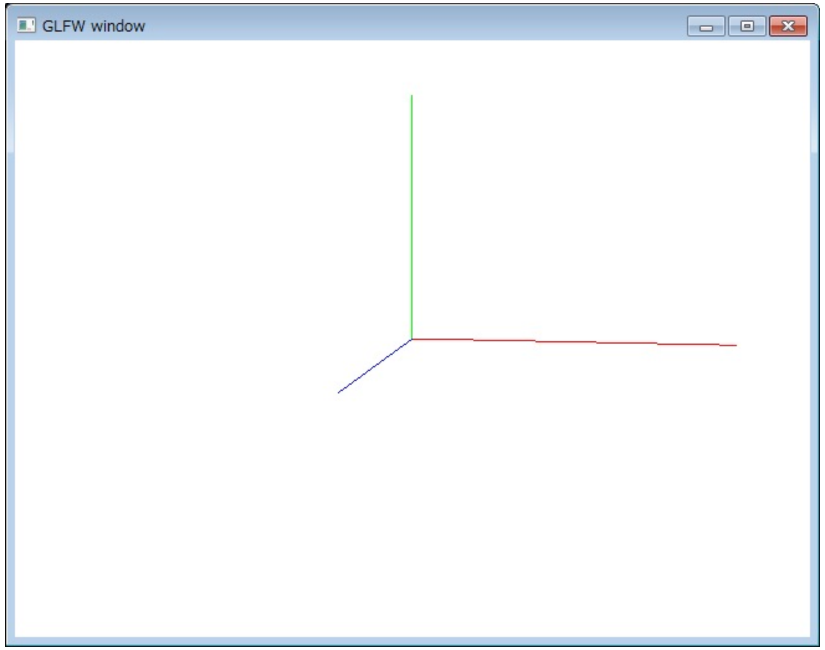
Viewport transform

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(
  45.0,         // field of view
  640 / 480,    // aspect ratio
  0.1, 100.0); // depth range
```

Projection transform

```
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
```

Modelview transform



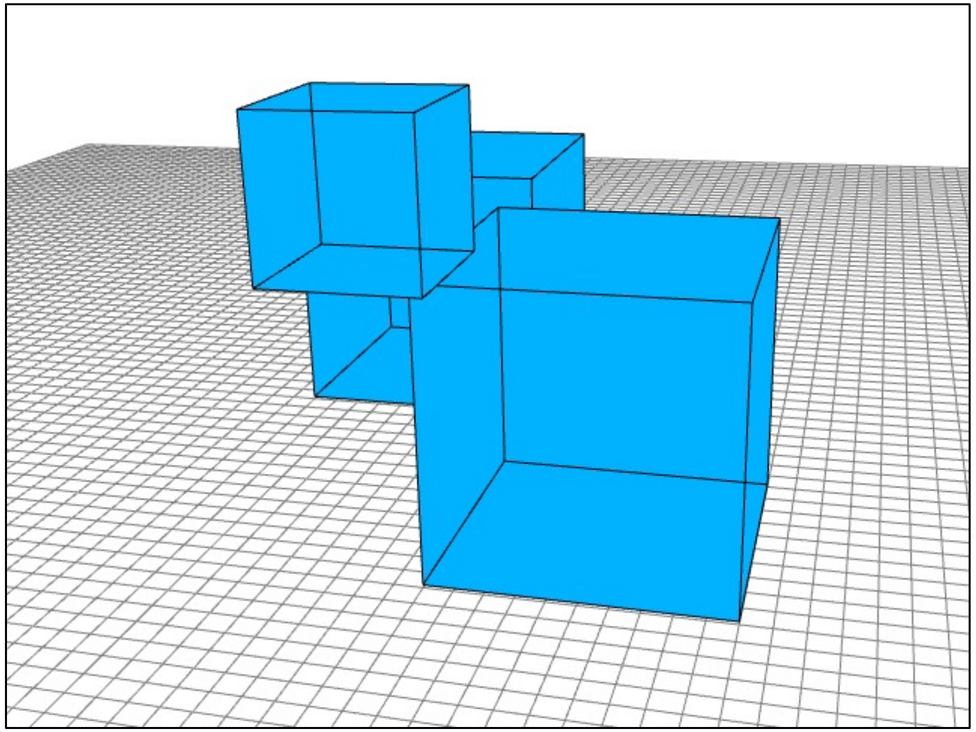
Output

```
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();
```

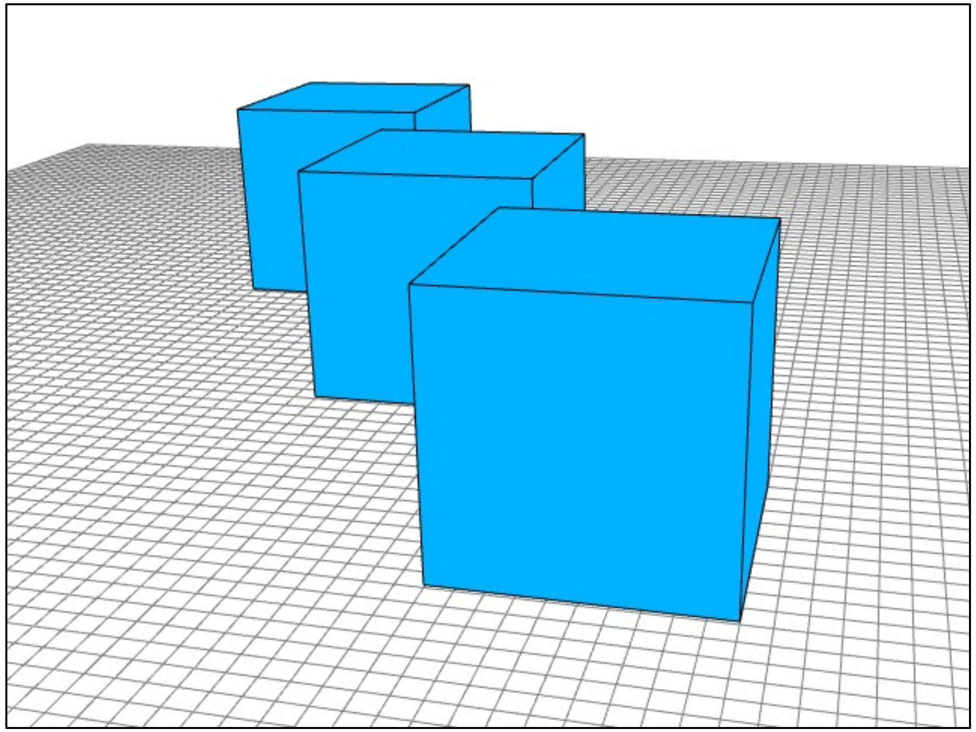
Scene content

Z-buffering

Hidden surface removal



Without hidden surface removal

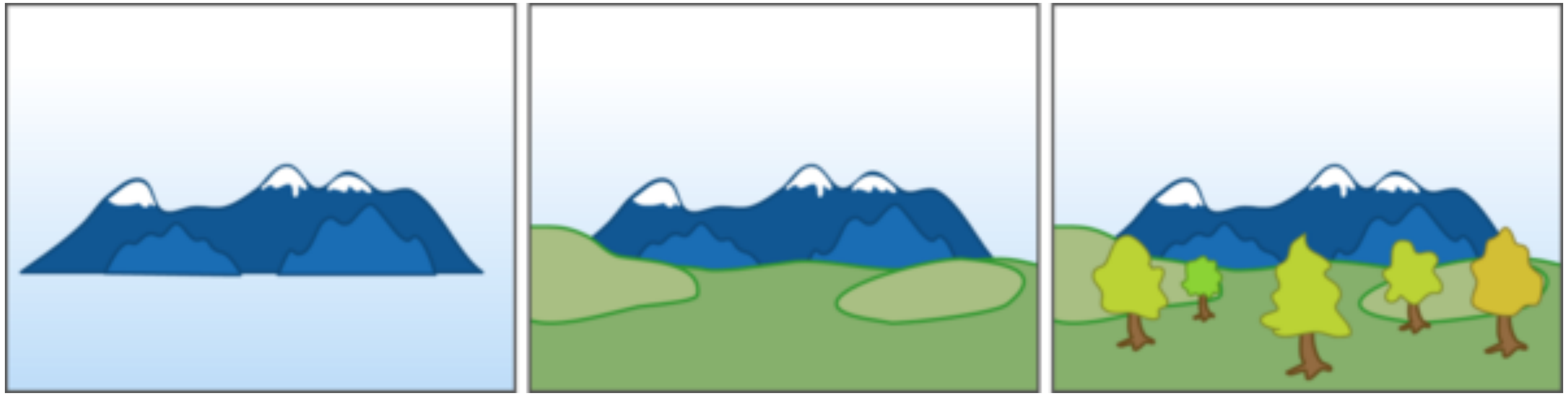


With hidden surface removal

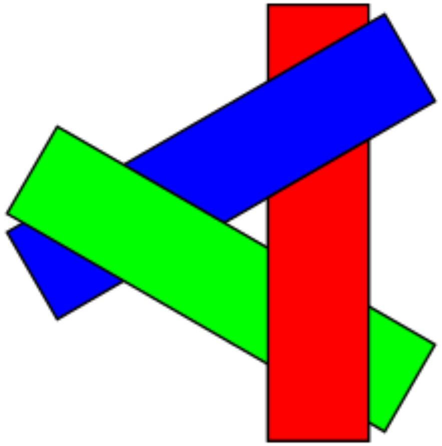
- Classic problem in CG

Painter's algorithm

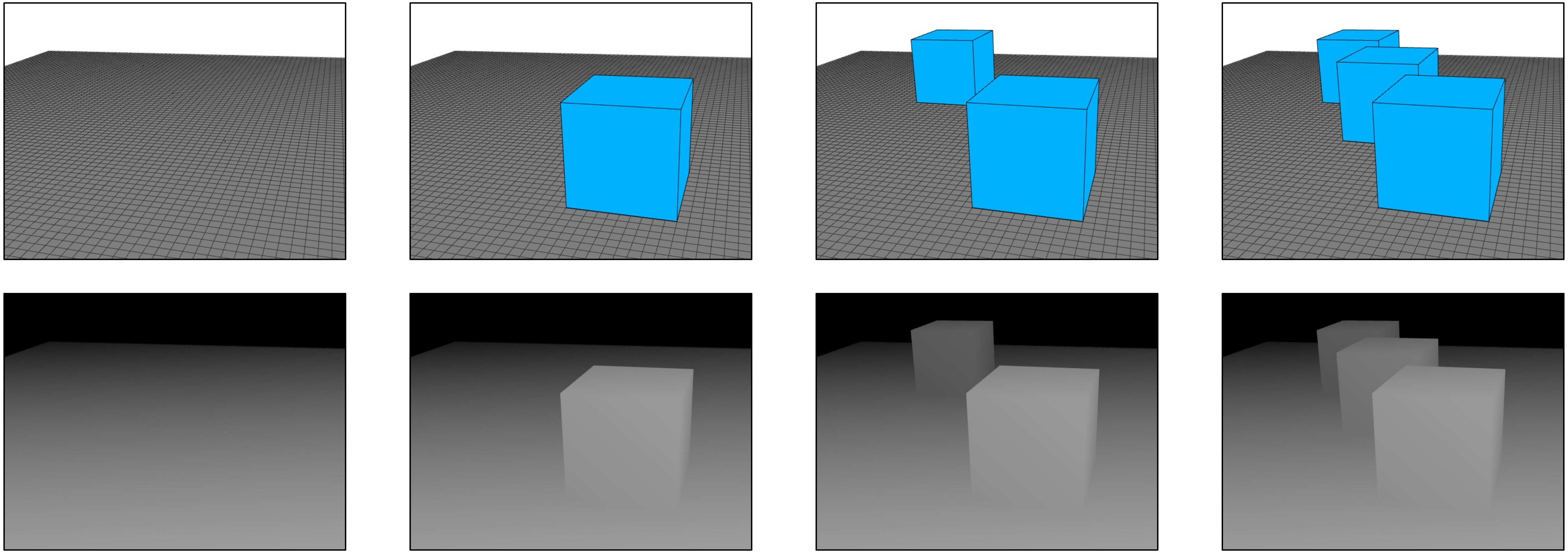
- Sort objects according to distances to camera, then draw them in the back-to-front order



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



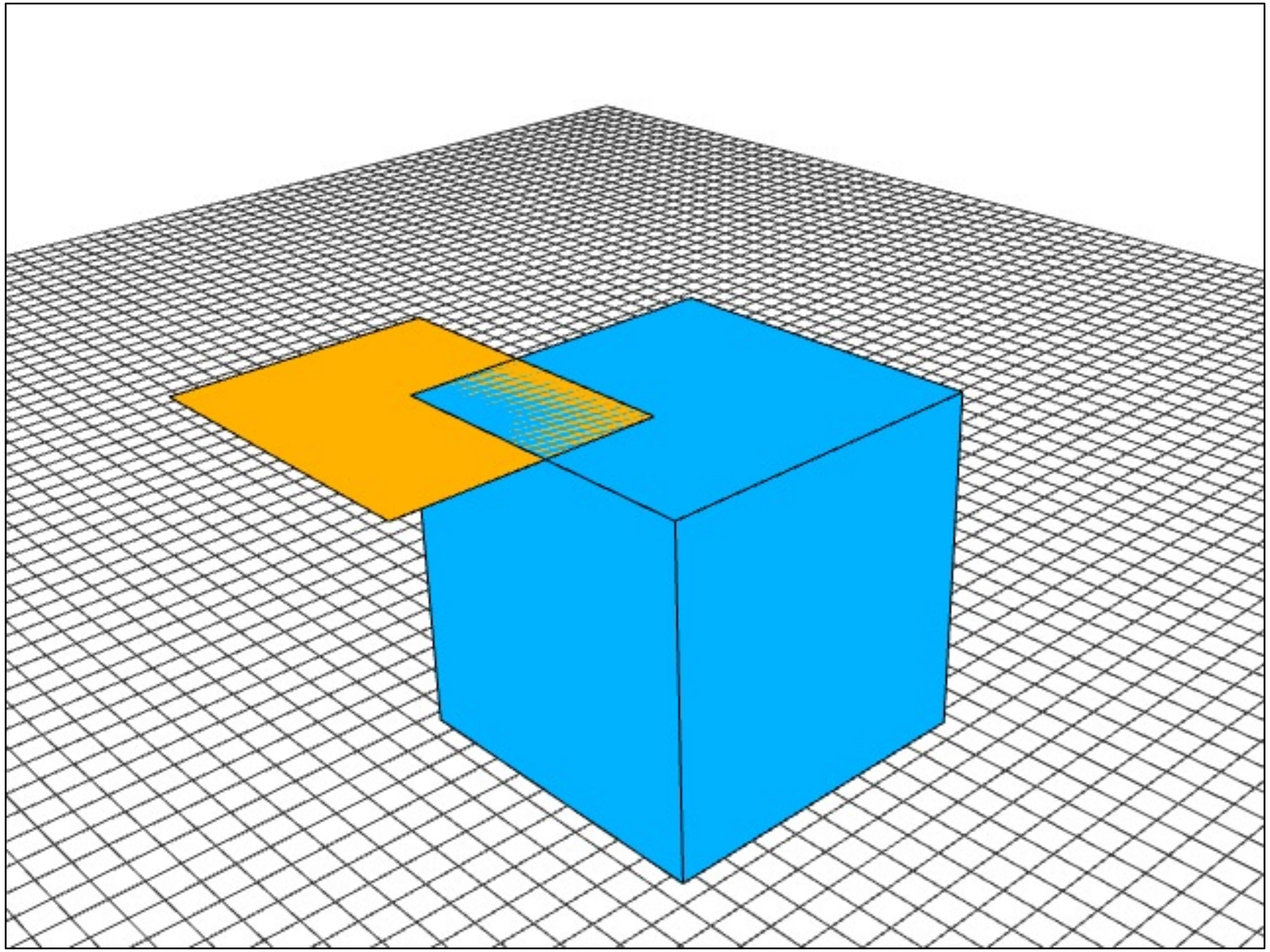
Z-buffering



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

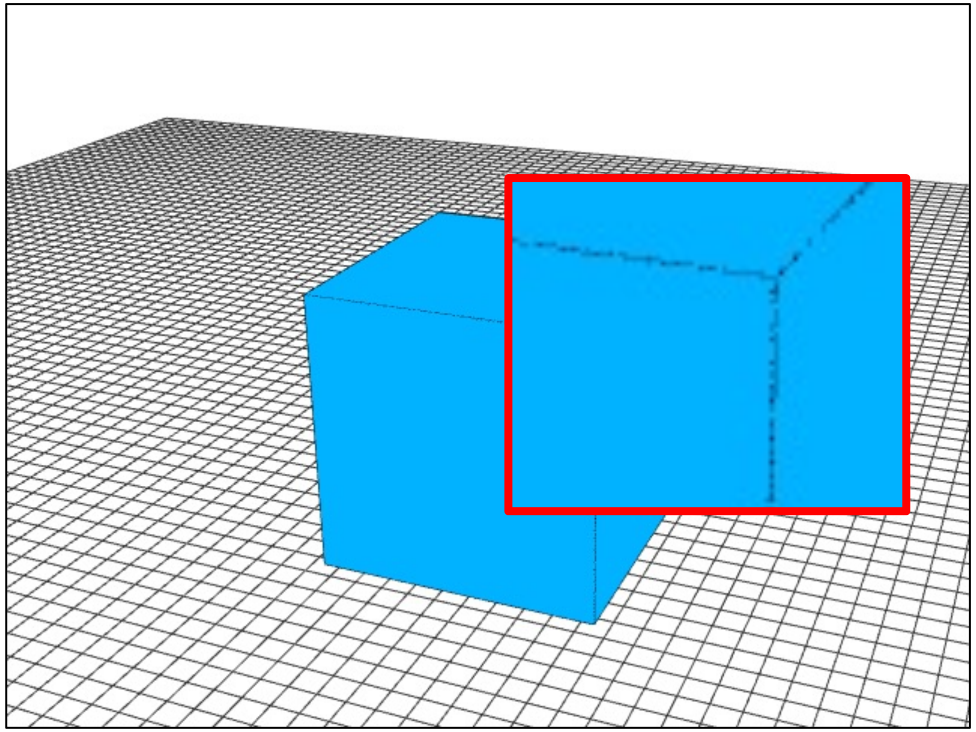
Typical issues with Z-buffering: Z-fighting

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

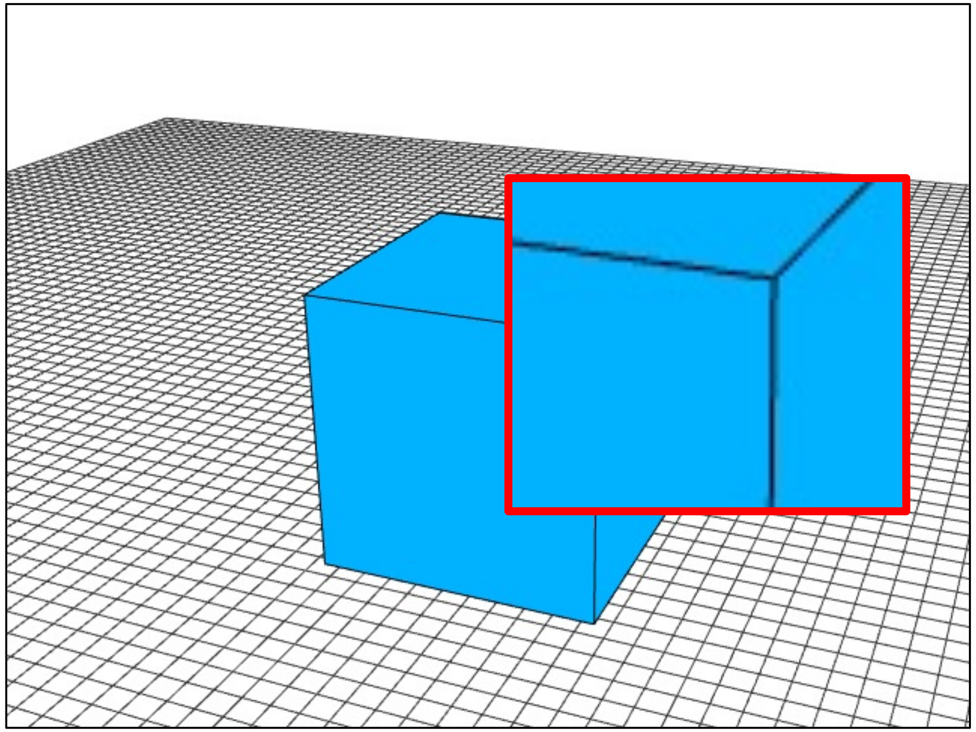


Typical issues with Z-buffering: Simultaneous drawing of faces and lines

- Dedicated OpenGL trick: glPolygonOffset



Without polygon offset

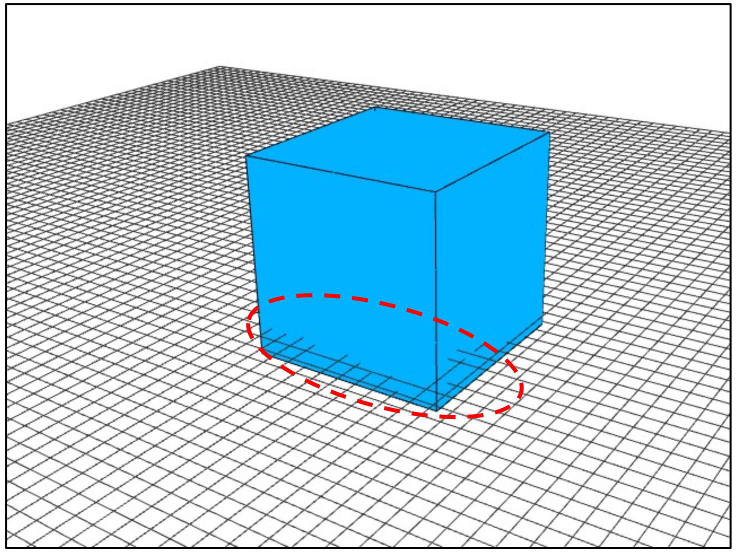


With polygon offset

Typical issues with Z-buffering: Depth range

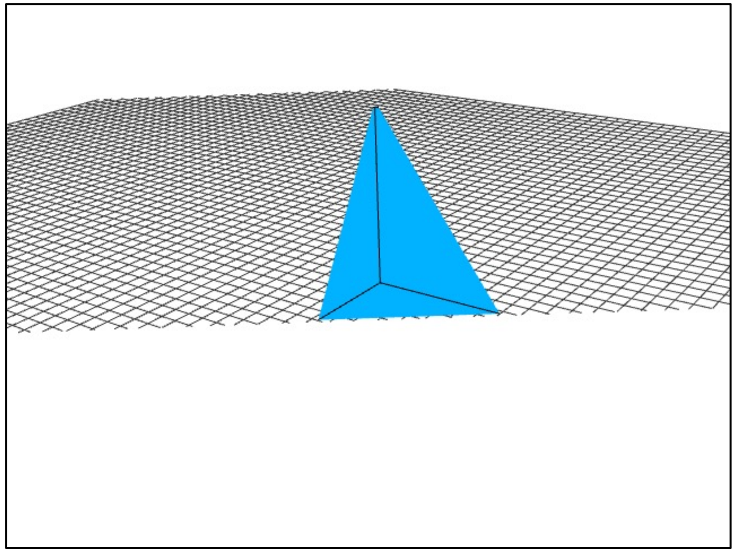
```
gluPerspective(
  45.0,           // field of view
  640 / 480,      // aspect ratio
  0.1, 1000.0); // zNear, zFar
```

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



zNear=0.0001
zFar =1000

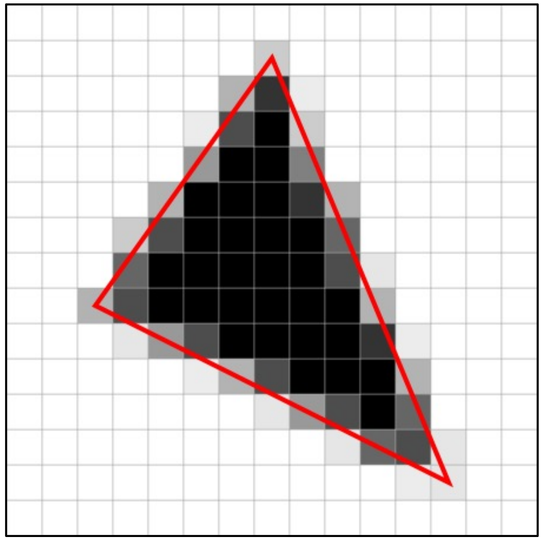
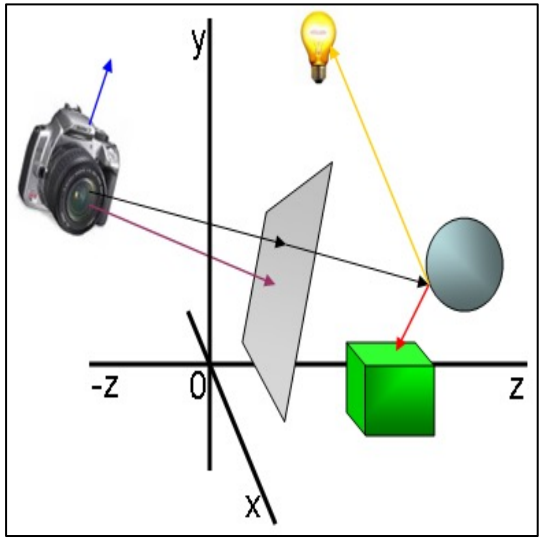
- Larger depth range
  - ➔ Larger drawing space, less accuracy



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	Per-pixel (ray) processing
	 One polygon updates multiple pixels	 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

(u_x, u_y, u_z) : axis vector

- Problems with matrix representation
  - Overly complex!
    - Should be represented by 2 DoF (axis direction) + 1 DoF (angle) = 3 DoF
  - Can't handle interpolation (blending) well

Degree of Freedom

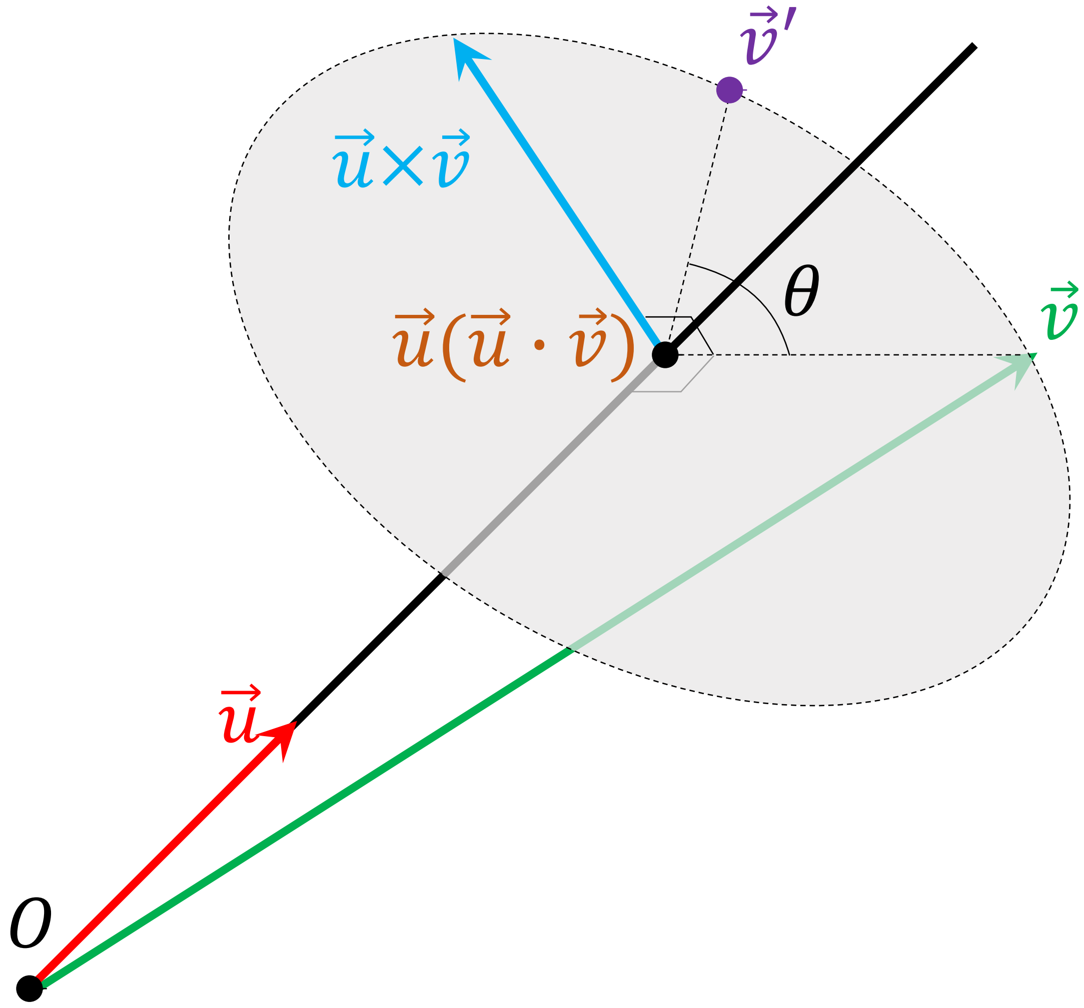
Geometry of axis-angle rotation

$\vec{u}$: 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
  - $\mathbf{i}^2 = -1$
  - $\mathbf{c} = (a, b) \coloneqq a + b\,\mathbf{i}$
  - $\mathbf{c}_1\mathbf{c}_2 = (a_1, b_1)(a_2, b_2) = a_1a_2 - b_1b_2 + (a_1b_2 + b_1a_2)\,\mathbf{i}$

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

$$= (a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2) + (a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2)\,\mathbf{i}$$

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

Notation by scalar + 3D vector

- $\mathbf{q}_1 = a_1 + b_1\,\mathbf{i} + c_1\,\mathbf{j} + d_1\,\mathbf{k} \coloneqq a_1 + (b_1, c_1, d_1) = a_1 + \overrightarrow{v_1}$
- $\mathbf{q}_2 = a_2 + b_2\,\mathbf{i} + c_2\,\mathbf{j} + d_2\,\mathbf{k} \coloneqq a_2 + (b_2, c_2, d_2) = a_2 + \overrightarrow{v_2}$
- $$\begin{aligned}\mathbf{q}_1\mathbf{q}_2 = &(a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2) + \\ &(a_1b_2 + a_2b_1 + c_1d_2 - d_1c_2)\,\mathbf{i} + \\ &(a_1c_2 + a_2c_1 + d_1b_2 - b_1d_2)\,\mathbf{j} + \\ &(a_1d_2 + a_2d_1 + b_1c_2 - c_1b_2)\,\mathbf{k}\end{aligned}$$

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

Rotation using quaternions

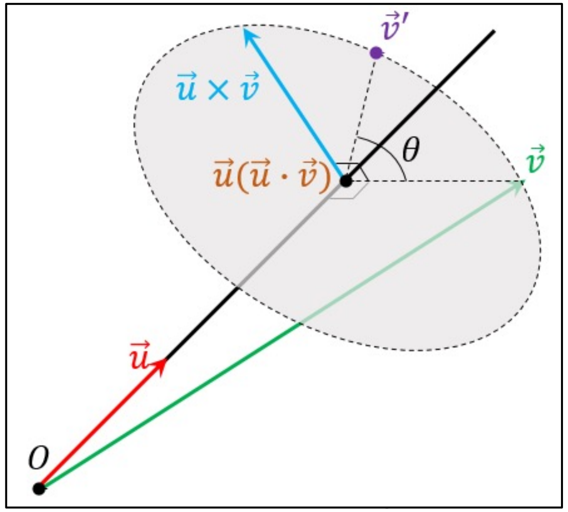
$$q = \cos\frac{\alpha}{2} + \vec{u}\sin\frac{\alpha}{2}$$

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

$$\vec{v}' = q\vec{v}q^{-1} = \left(\cos\frac{\alpha}{2} + \vec{u}\sin\frac{\alpha}{2}\right)\vec{v}\left(\cos\frac{\alpha}{2} - \vec{u}\sin\frac{\alpha}{2}\right)$$

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

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

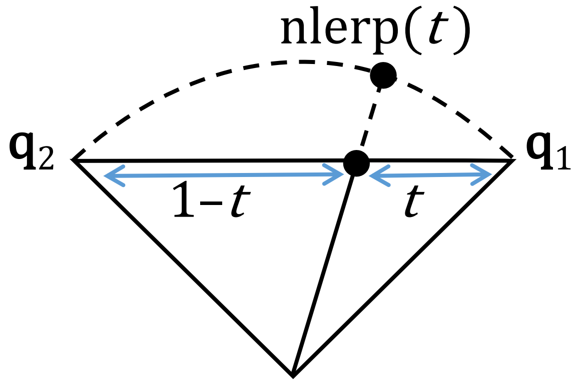


- Interesting theory behind
  - Clifford algebra
  - Geometric algebra
- Also important for physics & robotics

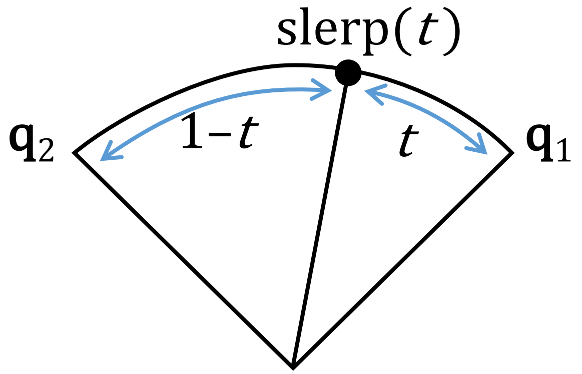
https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation

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



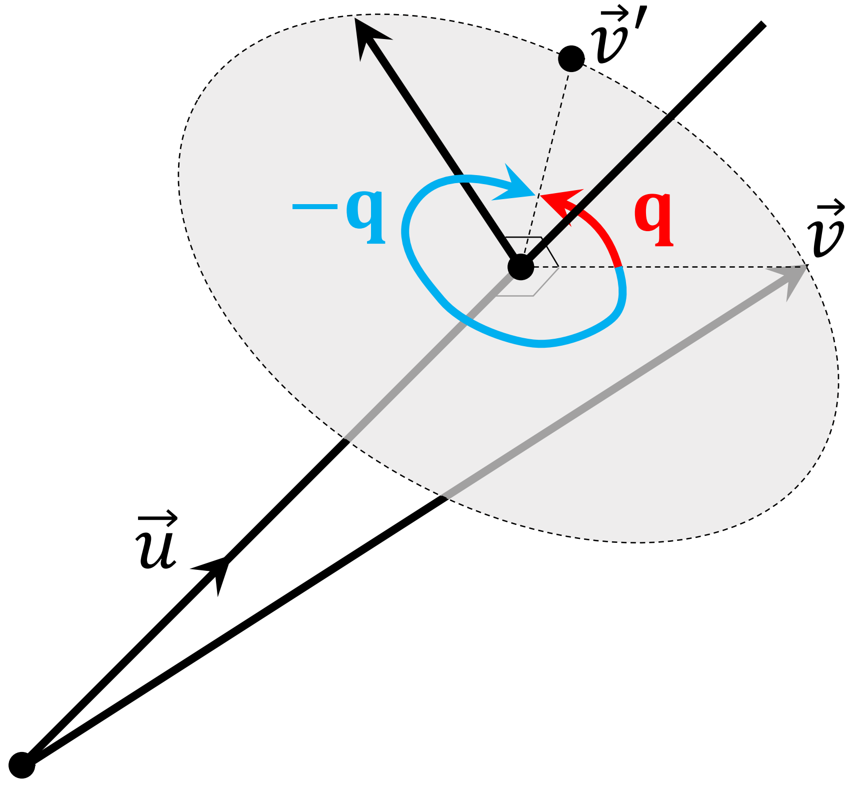
- Spherical linear interpolation (slerp)
  - $\Omega = \cos^{-1}(\mathbf{q}_1 \cdot \mathbf{q}_2)$
  - $\mathrm{slerp}(\mathbf{q}_1, \mathbf{q}_2, t) \coloneqq \frac{\sin(1-t)\Omega}{\sin\Omega}\mathbf{q}_1 + \frac{\sin t\Omega}{\sin\Omega}\mathbf{q}_2$
  - ☹more computation, ☺constant angular speed



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

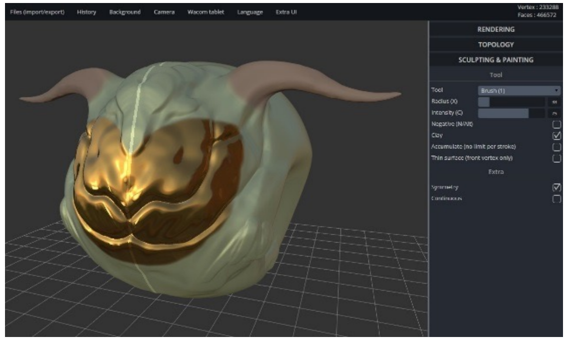
- Two kinds of APIs for using GPU
  - Different API designs (slightly?)
  - Both supported by most popular programming languages

- 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();
}
```

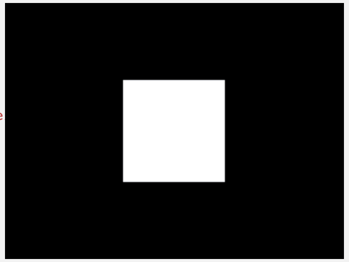
http://wisdom.sakura.ne.jp/system/opengl/gl20.html

WebGL

```html
<html>
  <head>
    <title>WebGL Demo</title>
    <script src="gl-matrix-min.js"></script>
    <script>
main();
function main() {
  const canvas = document.querySelector('#glcanvas');
  const gl = canvas.getContext('webgl');
  if (!gl) {
    alert('Unable to initialize WebGL!');
    return;
  }
  const vsSource = `
    attribute vec4 aVertexPosition;
    uniform mat4 uModelViewMatrix;
    uniform ma
    void main(
      gl_Posit
    }
  `;
  const fsSour
    void main(
      gl_Frag
    }
  `;
  const shader
  const progra
    program: s
    attribLoca
      vertexPo
    },
    uniformLoc
      projecti
      modelVie
    },
  };
  const buffer
  drawScene(gl
}
function initB
  const positi
  gl.bindBuffe
  const positi
     1.0,  1.0
    -1.0,  1.0
     1.0, -1.0
    -1.0, -1.0
  ];
  gl.bufferDat
  return {
    position:
  };
}
function drawS
  gl.clearColo
  gl.clearDept
  gl.enable(gl
  gl.depthFunc
  gl.clear(gl.
  const fieldO
  const aspect = gl.canvas.clientWidth / gl.canvas.clientHeight;
  const zNear = 0.1;
  const zFar = 100.0;
  const projectionMatrix = mat4.create();
  mat4.perspective(projectionMatrix, fieldOfView, aspect, zNear, zFar);
  const modelViewMatrix = mat4.create();
  mat4.translate(modelViewMatrix, modelViewMatrix, [-0.0, 0.0, -6.0]);
```

```javascript
  {
    const numComponents = 2;
    const type = gl.FLOAT;
    const normalize = false;
    const stride = 0;
    const offset = 0;
    gl.bindBuffer(gl.ARRAY_BUFFER, buffers.position);
    gl.vertexAttribPointer(
      programInfo.attribLocations.vertexPosition,
      numComponents,
      type,
      normalize,
      stride,
      offset);
    gl.enableVertexAttribArray(programInfo.attribLocations.vertexPosition);
  }
m(programInfo.program);
trix4fv(
nfo.uniformLocations.projectionMatrix,

onMatrix);
trix4fv(
nfo.uniformLocations.modelViewMatrix,

wMatrix);

et = 0;
exCount = 4;
ays(gl.TRIANGLE_STRIP, offset, vertexCount);

haderProgram(gl, vsSource, fsSource) {
Shader = loadShader(gl, gl.VERTEX_SHADER, vsSource);
ntShader = loadShader(gl, gl.FRAGMENT_SHADER, fsSource);
Program = gl.createProgram();
der(shaderProgram, vertexShader);
der(shaderProgram, fragmentShader);
am(shaderProgram);
rogramParameter(shaderProgram, gl.LINK_STATUS)) {
ble to initialize the shader program: '
ProgramInfoLog(shaderProgram));
l;

rProgram;

hader(gl, type, source) {
 = gl.createShader(type);
rce(shader, source);
ader(shader);
haderParameter(shader, gl.COMPILE_STATUS)) {
error occurred compiling the shaders: '
ShaderInfoLog(shader));
hader(shader);
l;

r;

="glcanvas" width="640" he
  </body>
</html>
```

```javascript
  gl.bindBuffer(gl.ARRAY_BUFFER, buffers.positi
  gl.vertexAttribPointer(
    programInfo.attribLocations.vertexPosition,
    numComponents,
    type,
    normalize,
    stride,
    offset);
  gl.enableVertexAttribArray(programInfo.attrib
}
gl.useProgram(programInfo.program);
gl.uniformMatrix4fv(
    programInfo.uniformLocations.projectionMatr
    false,
    projectionMatrix);
gl.uniformMatrix4fv(
    programInfo.uniformLocations.modelViewMatri
    false,
    modelViewMatrix);
{
  const offset = 0;
  const vertexCount = 4;
  gl.drawArrays(gl.TRIANGLE_STRIP, offset, vert
```



https://mdn.github.io/webgl-examples/tutorial/sample2/

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

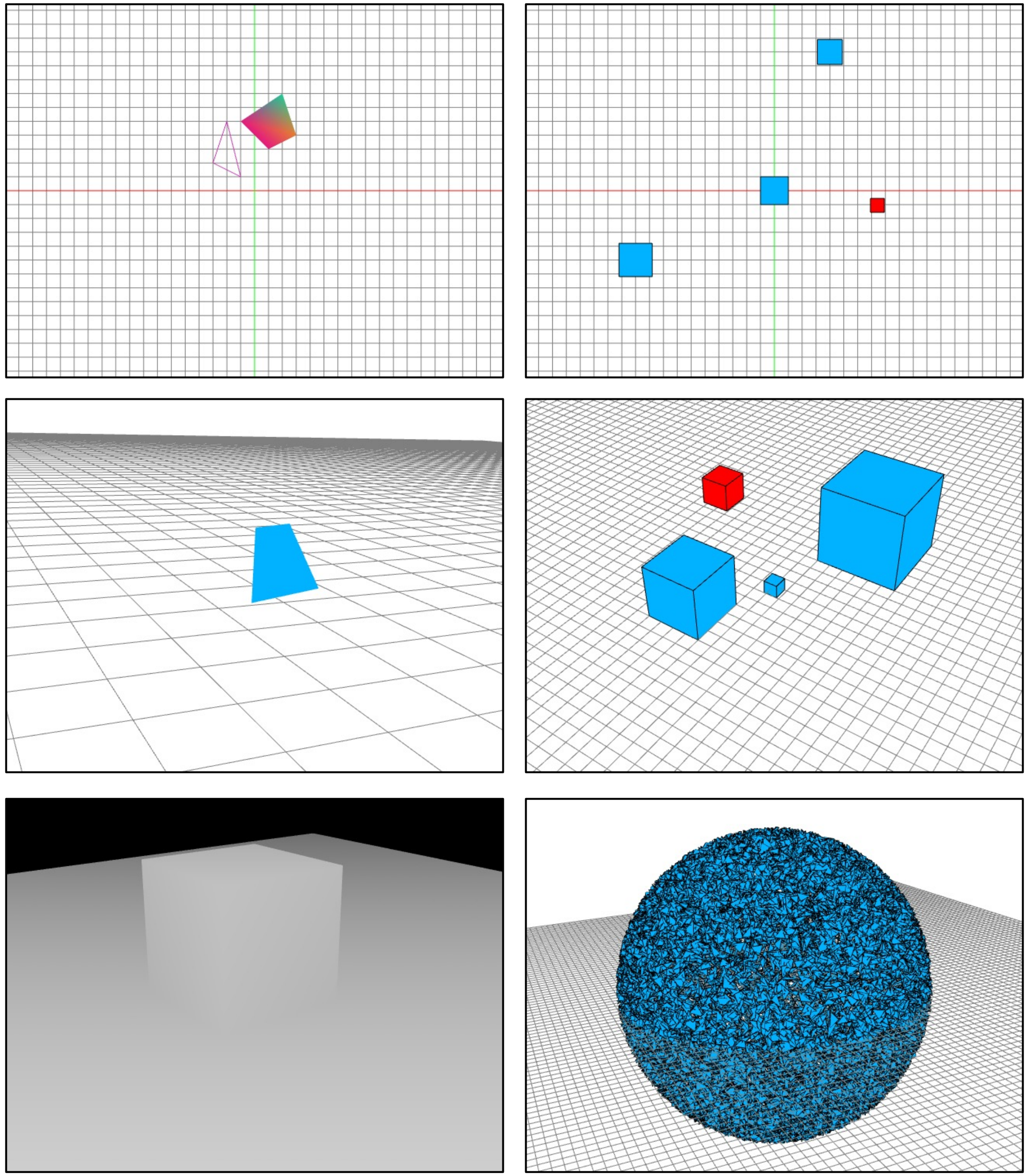
```
<script src="js/three.min.js"></script>
<script>
var camera, scene, renderer, geometry, material, mesh;
function init() {
  scene = new THREE.Scene();
  camera = new THREE.PerspectiveCamera( 75, 640 / 480, 1, 10000 );
  camera.position.z = 1000;
  geometry = new THREE.BoxGeometry( 200, 200, 200 );
  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

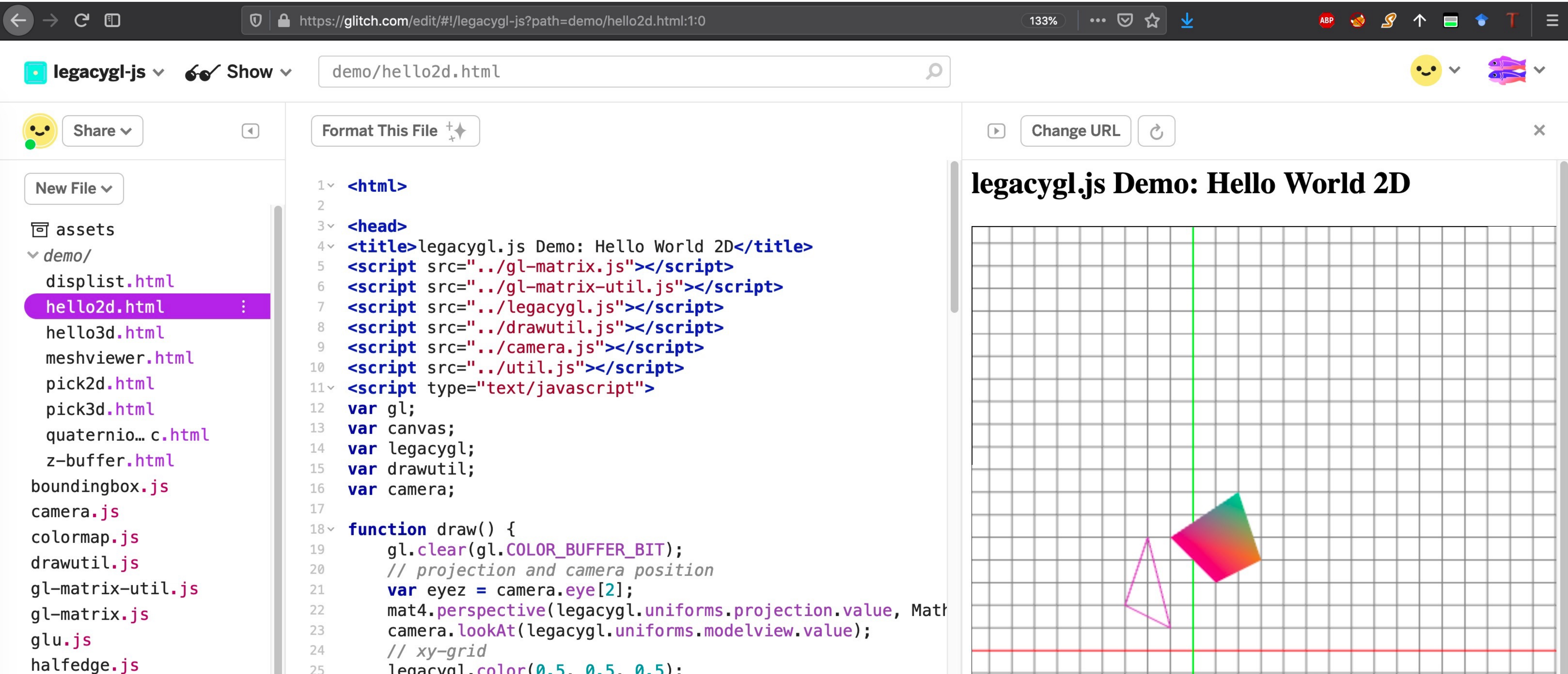
- Developed by me for this course
  - https://bitbucket.org/kenshi84/legacygl.js
  - Demos & tutorial

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



WebGL development using Glitch

- A free web space for putting js/html/css https://glitch.com/
- Online editing and quick previewing



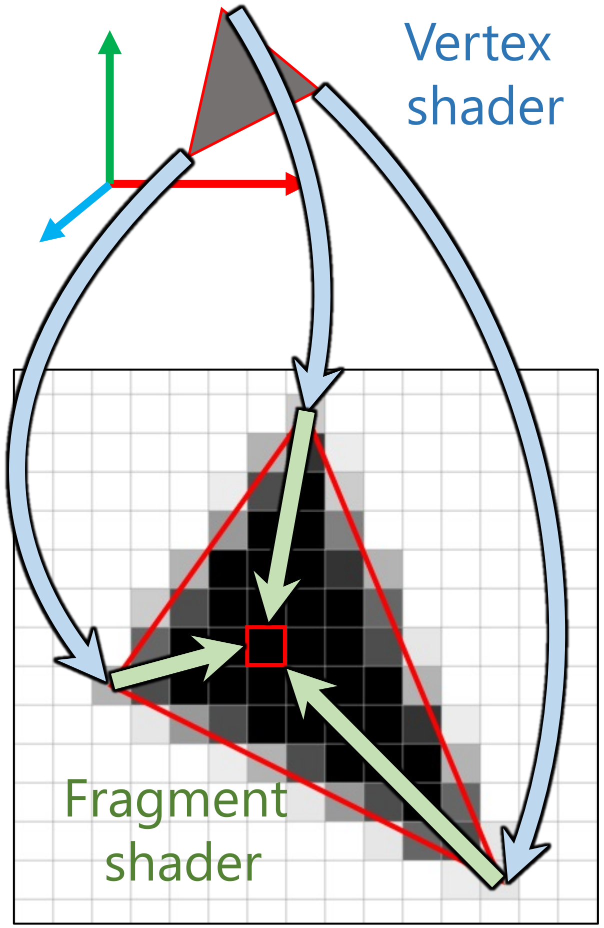
How to work on assignemnts

- Implement your solution using WebGL, upload it to the web, submit the URL via LMS
  - Glitch is recommended, but other means (e.g. GitHub Pages, your own server) is also OK
  - Include some descriptions/discussions/etc in the HTML page
  - OK to use other WebGL libraries (e.g. three.js)

- Other programming languages (e.g. C++) are also allowed
  - Should compile and run on typical computing systems
  - Include source+binary+doc in a single ZIP file, upload it somewhere (e.g. Google Drive), submit its URL

- If you have any questions, don't hesitate to 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

(Grammar differs slightly across 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

https://www.khronos.org/registry/OpenGL/specs/es/2.0/GLSL_ES_Specification_1.00.pdf

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/
  - Mozilla Developer Network
    - https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API
  - 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