

Today's Agenda

Basic design of a graphics system

Introduction to OpenGL

Image Compositing

Compositing one image over another is most common choice

- can think of each image drawn on a transparent plastic sheet
- the final image is formed by stacking layers together

Given images **A** & **B**, we can compute **C = A over B**

$$C_{rgb} = \alpha_A A_{rgb} + (1 - \alpha_A) \alpha_B B_{rgb}$$

- if we pre-multiply α values, this simplifies to

$$C' = A' + (1 - \alpha_A) B'$$

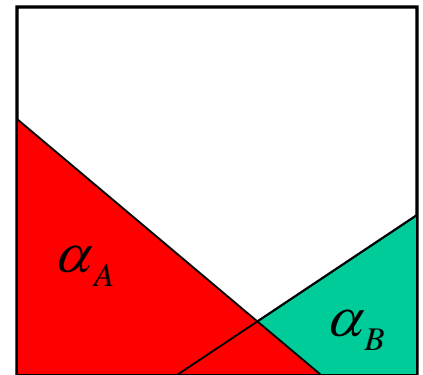


Image Formation

Elements of image formation:

- Illumination sources
- Objects
- Viewer (e.g., camera and eye)
- Attributes of materials

How can we design graphics hardware and software to mimic the image formation process?

Practical Approach

Process objects one at a time in the order they are generated by the application

Pipeline architecture



All steps can be implemented in hardware on the graphics card

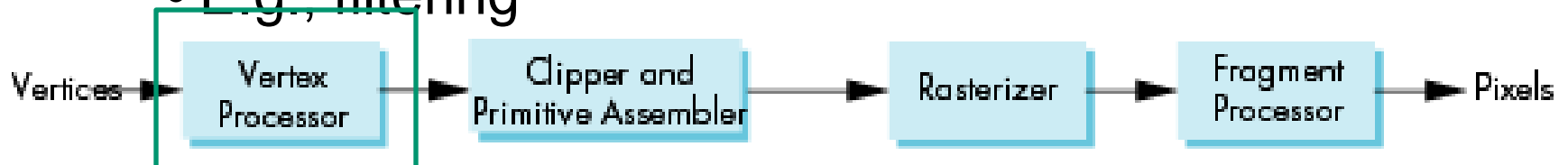
Vertex Processing

Geometrical transformation:

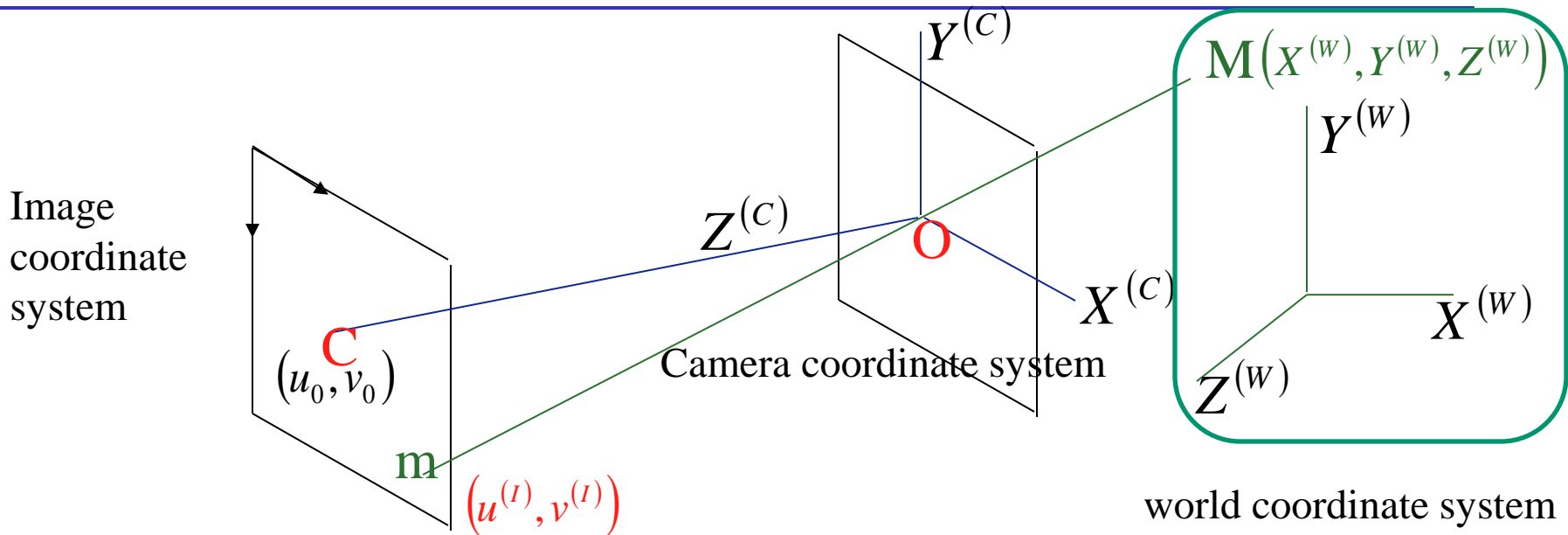
- Convert object representations from one coordinate system to another
 - World (local) coordinate system built on each object
 - Camera (eye) coordinate system
 - Screen/image coordinate system
- Changing coordinates is computed by matrix-vector multiplication

Color transformations: Vertex processor also computes vertex colors

- E.g., filtering



Vertex Processing Example: Perspective Projection Geometry




How a 3D point measured in world coordinate system produce an image point measured in the image plane/coordinate system?

How to Project a 3D Point in Camera Coordinate System to a 2D Point in Row-Column Image Frame

Spatial resolution & image center
Relative position between object & camera

$$\lambda \begin{bmatrix} u^{(I)} \\ v^{(I)} \\ 1 \end{bmatrix} = \begin{bmatrix} k_u & 0 & u_0 \\ 0 & k_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -f & 0 & 0 & 0 \\ 0 & -f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R_{3 \times 3} & T_{3 \times 1} \\ 0_{1 \times 3} & 1 \end{bmatrix} \begin{bmatrix} X^{(W)} \\ Y^{(W)} \\ Z^{(W)} \\ 1 \end{bmatrix}$$

2D Image point
Focus length
3D object point

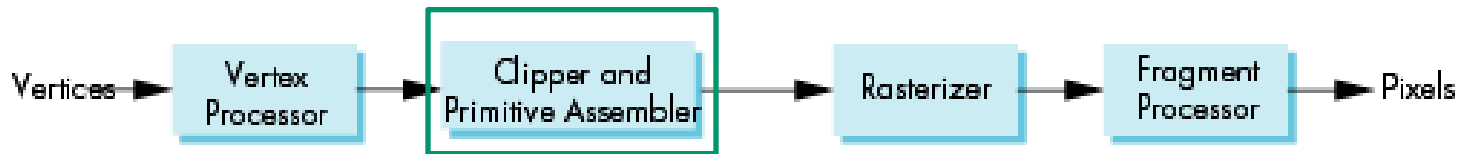


 $P_{3 \times 4}$

Primitive Assembly

Vertices must be collected into geometric objects before clipping and rasterization

- Line segments
- Polygons
- Curves and surfaces

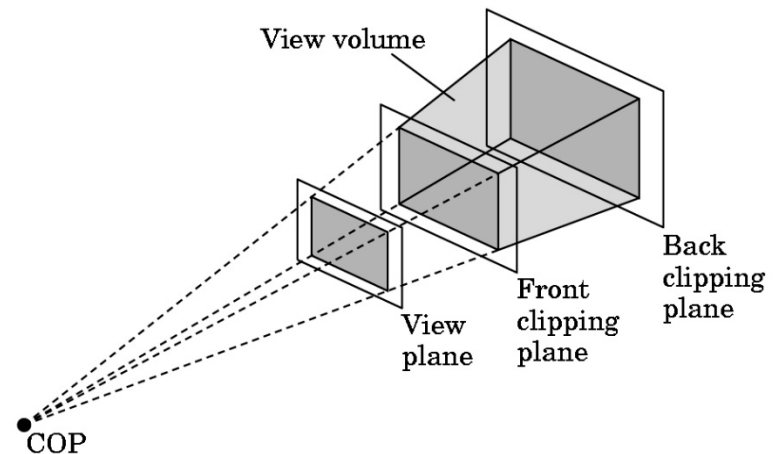
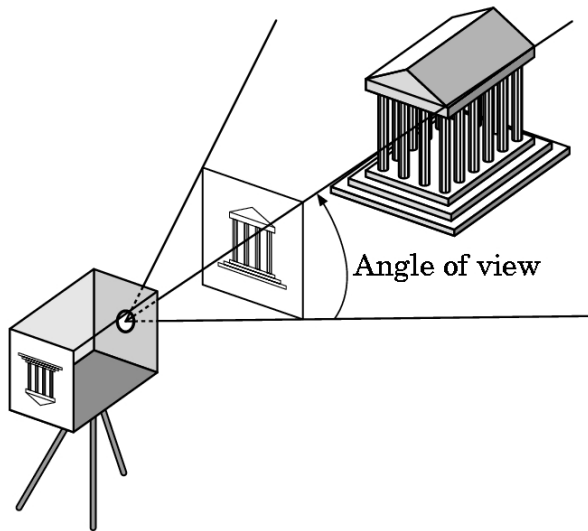


E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

Clipping

The virtual camera can only see part of the world or object space according to the field of view (angle of view)

- Objects that are not within this volume are said to be *clipped* out of the scene



Rasterization

Vertices → Pixels

For the objects in the clipping volume, the pixels in the frame buffer must be assigned colors

Rasterizer produces a set of fragments for each object

Fragments are “potential pixels”

- Have a location in frame buffer
- Color and **depth attributes**



Fragment Processing

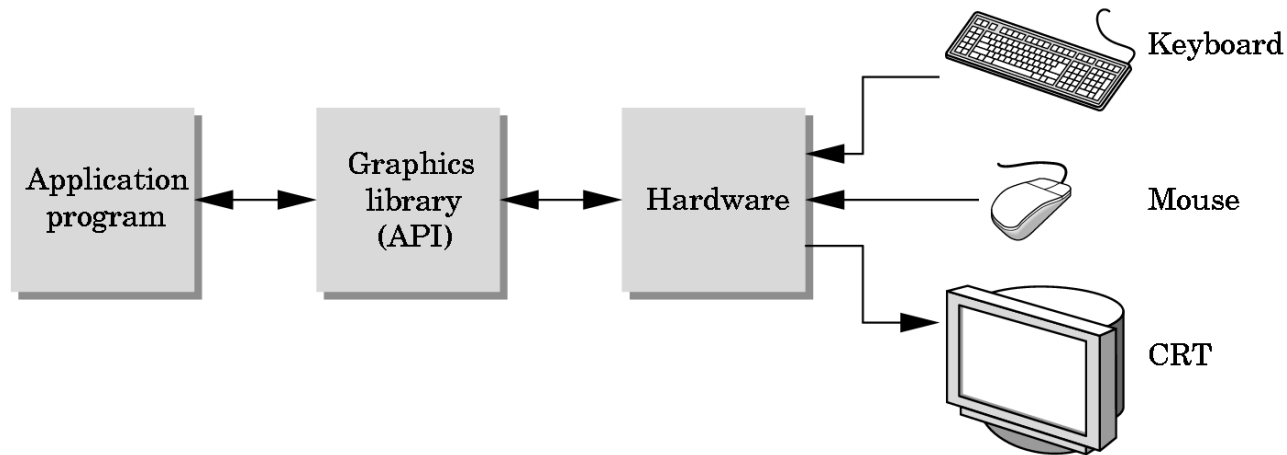
Fragments are processed to determine the color of the corresponding pixel in the frame buffer

Colors can be determined by texture mapping or interpolation of vertex colors



The Programmer's Interface

Programmer sees the graphics system through a software interface: the Application Programmer Interface (API)



API Contents

Functions that specify what we need to form an image

- Objects
- Viewer
- Light Source(s)
- Materials

Other information

- Input from devices such as mouse and keyboard

Object Specification

Most APIs support a limited set of primitives including

- Points (0D object)
- Line segments (1D objects)
- Polygons (2D objects)
- Some curves and surfaces
 - Quadrics
 - Parametric polynomials

All are defined through locations in space or *vertices*

Introduction to OpenGL

OpenGL is an Application Programmer Interface (API) and a standard graphics library for 2-D & 3-D drawing

- maps fairly directly to graphics hardware
- doesn't address windows or input events (we'll use GLUT)
- platform-independent
- a free software implementation (www.mesa3d.org)

OpenGL 3.1 Totally shader-based

- Each application must provide both a vertex and a fragment shader

OpenGL 4.1 and 4.2

- Add geometry shaders and tessellator

OpenGL Libraries

OpenGL core library

- OpenGL on Windows
- GL on most unix/linux systems (libGL.a)

Available when you install the graphics driver

Examples: OpenGL library calls

- glBegin()
- glEnd()
- glVertex()
- glColor()
- glClear()
- glDrawPixels()
- glLight() and more...

OpenGL Libraries

OpenGL Utility Library (GLU)

- Provides functionality in OpenGL core,

Examples: GLU library calls

- gluBeginSurface()
- gluEndSurface()
- gluSphere()
- gluCylinder()
- gluOrtho2D()
- gluPerspective()
- gluLookAt() and more...

OpenGL Libraries

OpenGL Utility Toolkit (GLUT/FreeGLUT)

- Provides functionality for all window systems
 - creating windows,
 - receiving inputs,
 - handling events, etc.
- Some functionality can't work since it requires deprecated functions
- FreeGLUT contains the latest developments

Examples: GLUT library calls

- `glutCreateWindow()`
- `glutInit()`
- `glutDisplayFunc()`
- `glutMouseFunc()`
- `glutKeyboardFunc()`
- `glutMainLoop()` and more...

OpenGL Extension Libraries

Links with window system

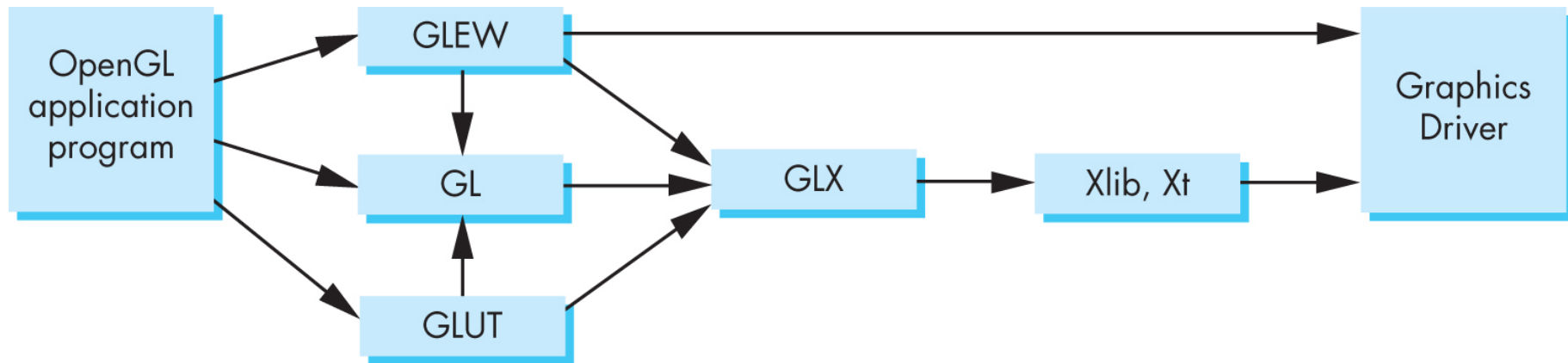
- GLX for X window systems
- WGL for Windows
- AGL for Macintosh

OpenGL Extension Wrangler Library (GLEW)

<http://glew.sourceforge.net/>

- Need to be initialized after creating windows
- Verify OpenGL extensions on a specific platform
- Deal with function pointers

Software Organization



OpenGL Functions

Primitives – low level objects that can be displayed

- Points, line Segments, and triangles

Attributes - how objects are displayed

- Color, pattern, etc.

Transformations

- Viewing
- Modeling

Control (GLUT)

- Initialization, windows, etc.

Input (GLUT)

Query

- Timer, occlusion, primitives, etc.

OpenGL State

OpenGL is a state machine

OpenGL functions are of two types

- Primitive generating
 - Very few
 - Can cause output if primitive is visible
 - How vertices are processed and appearance of primitive are controlled by the state
- State changing
 - Transformation functions
 - Attribute functions
 - Under 3.1 most state variables are defined by the application and sent to the shaders

Extensive State Information

Always a “current” state, initialized with default values

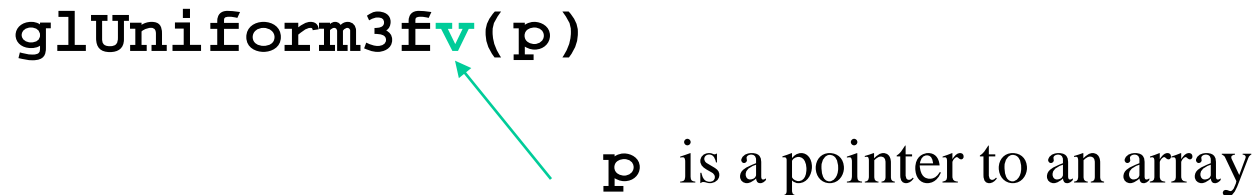
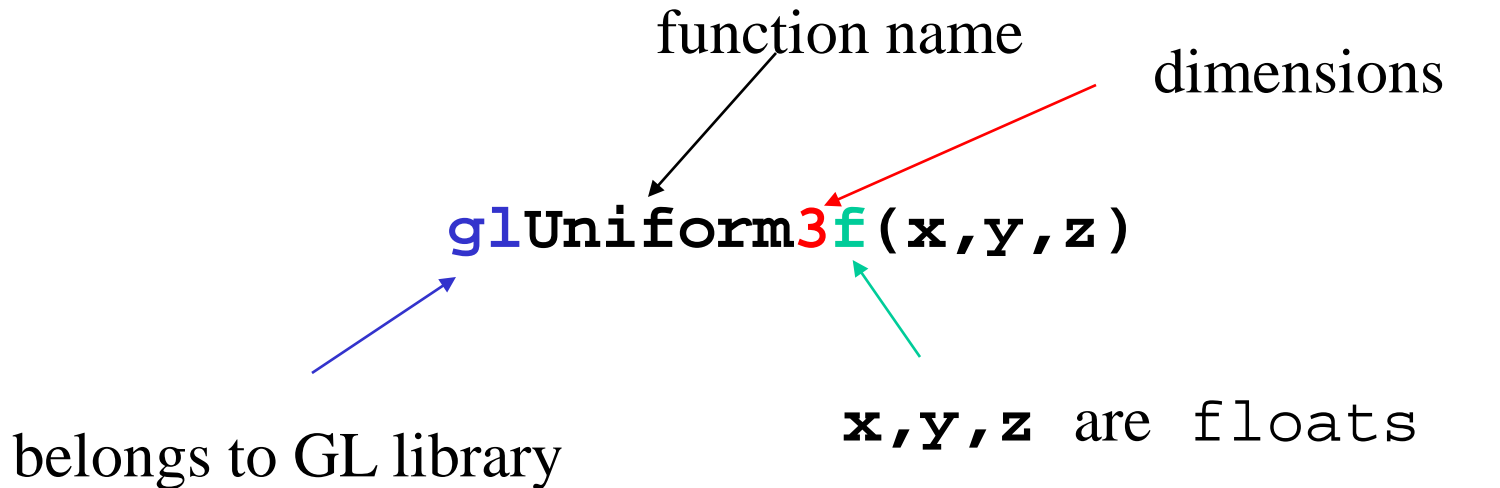
- changes to the state apply to all subsequent operations

```
set_color(Red);  
draw_triangle(t1); // red  
draw_triangle(t2); // red  
draw_triangle(t3); // red  
set_color(Blue);  
draw_triangle(t4); // blue
```

Examples of state information include

- current drawing color, line width, point size, ...
- coordinate system (defined by transformation matrix)
- enabled features: depth tests, alpha blending, lighting, ...

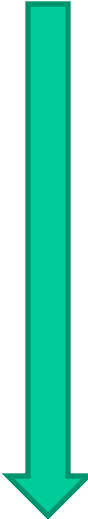
OpenGL function format



Shader-based OpenGL

Most state variables, attributes and related pre 3.1 OpenGL functions have been deprecated

OpenGL rendering pipeline (after 3.1) uses shaders in Vertex Processor

- 
- **Vertex shading stage:** receiving and process primitives separately
 - E.g., specifying the colors and positions
 - Tessellation shading stage: specifying a patch, i.e., an ordered list of vertices and generating a mesh of primitives
 - Geometry shading stage: enabling multivertex access, changing primitive type
 - **Fragment shading stage:** processing color and depth

GLSL

OpenGL Shading Language

Like a complete C program supporting

- Matrix and vector types (2, 3, 4 dimensional)
- Overloaded operators
- C++ like constructors

Code sent to shaders as source code

Entry point is the main function main()

OpenGL functions to compile, link and get information to shaders

Simple Vertex Shader

```
#version 400
in vec4 vPosition;
void main(void)
{
    gl_Position = vPosition;
}
```

The version of OpenGL

Specify this is an input from application

must link to variable in application

built in variable

Simple Fragment Program

#version 400 Specify this is an output to the application

out vec4 fColor;

void main(void)

{

fColor = vec4(1.0, 0.0, 0.0, 1.0);

}

Reading Assignments

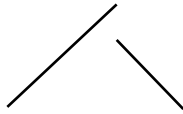
Chapter 2 of Angel

Chapter 1&2 of OpenGL Programming Guide

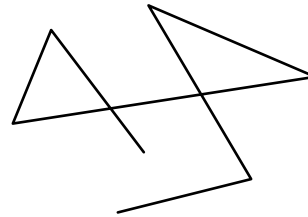
OpenGL Primitives



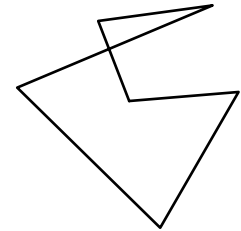
GL_POINTS



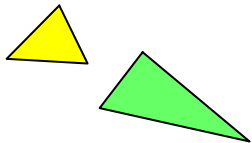
GL_LINES



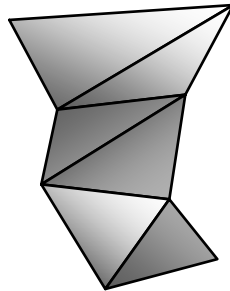
GL_LINE_STRIP



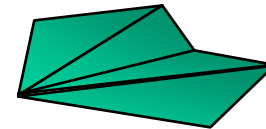
GL_LINE_LOOP



GL_TRIANGLES



GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN

Primitive #1: Points

Points are either 2- or 3-dimensional

- by convention, represent them as column vectors

$$\mathbf{v} = \begin{bmatrix} x \\ y \end{bmatrix} \quad \text{or} \quad \mathbf{v} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

A 2D point, a special case of a 3D point, can be represented as

- A 2D vector (e.g, `vec2(0,1)`),
- A 3D vector (e.g., `vec3(0,1,0)`),
- and more general a 4D vector (e.g., `vec4(0,1,0,1)`),

```
glDrawArrays(GL_POINTS, 0, N);
```