

# **Tutorial 1**

## **CS3241 Computer Graphics (AY22/23)**

*August 26, 2023*

**Wong Pei Xian**



[eo389023@u.nus.edu](mailto:eo389023@u.nus.edu)

## Question 1

To be able to display **realistic** images, our display devices need to be able to produce every frequency in the visible light spectrum.

True or false? Why? What are the advantages and disadvantages?

# Three-Color Theory

To be **realistic to human**

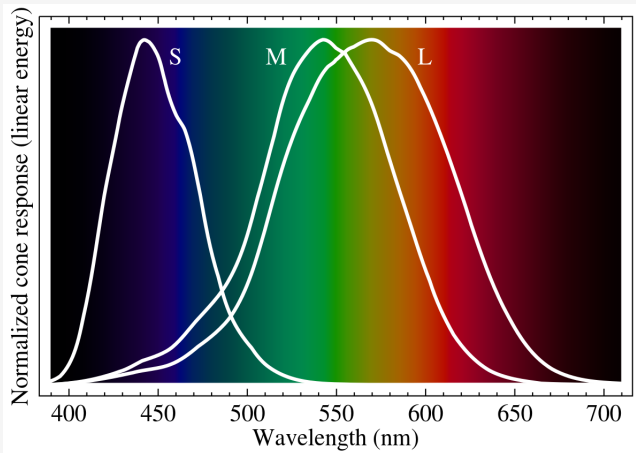
⇒ To be compatible with **human visual system** (Lo1, slide 35)

- Rods: Monochromatic
- **Cones**: Color sensitive to wavelengths
  - Long  $\approx$  **red**
  - Medium  $\approx$  **green**
  - Short  $\approx$  **blue**

**Proportion** of the three gives us the sensation of different colors.

# Cone sensitivity and Additive color theory

Single frequency = proportion of responses of each cone.



# Additive Color Display

Pros

1. We don't have to produce light of every wavelength in vis. light spectrum for realism
2. We can see colors that are **NOT** on vis. light spectrum (e.g. **PURPLE**).
  - Visible colors are different combinations of amount of activity in RGB cones
  - **Purple** is not a specific wavelength but a combination of red and blue wavelengths (unlike **violet** which has its own wavelength)

# Additive Color Display

Cons

Two different RGB values can produce the same color.

Q: What's an example of this?

A: There is no definitive inverse mapping of RGB to a wavelength, as it is display dependent.

<https://physics.stackexchange.com/questions/248139/can-two-different-rgb-color-triplets-give-the-same-color>

## Question 2

Each pixel in a frame-buffer has 8 bits for each of the R, G and B channels. How many different colors can each pixel represent? Is this enough?

8 bits per channel (R, G, B):  $2^{24} = \mathbf{16,777,216}$  colors.

# 32-bit color

In most graphical applications today, we have 4 channels:

- R
- G
- B
- (not included in this question) A for Alpha (transparency)

Hence color has 32 bits:  $2^{32}$  values (can be represented with an int)!

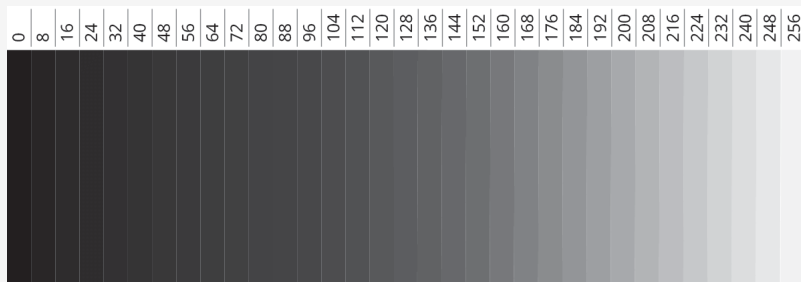
On color depth: <https://computer.howstuffworks.com/monitor4.htm>



# 32-bit color

Is this enough?

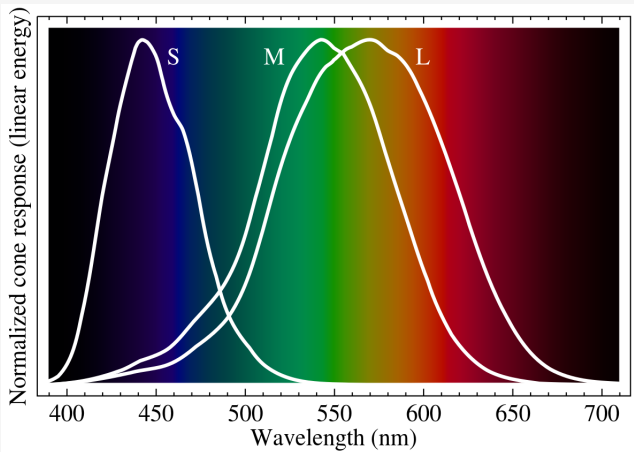
256 shades of gray: **banding artifacts.**



Use case decides if this is undesirable or not.

## 8-bit representation of color

On some systems, each pixel has only 8 bits (for all R, G, and B combined). How would you allocate the bits to the R, G and B primaries?



## 8-bit representation of color

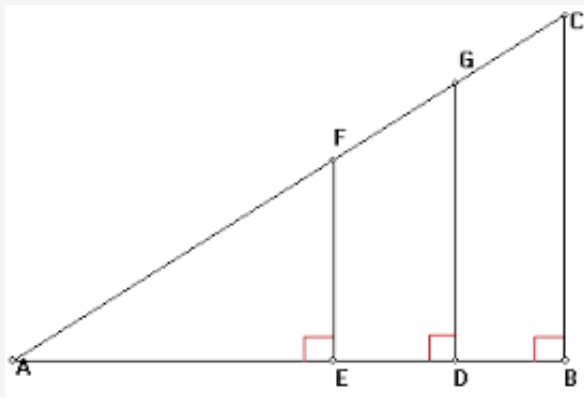
On some systems, each pixel has only 8 bits (for all R, G, and B combined). How would you allocate the bits to the R, G and B primaries?

3:3:2 for R:G:B. Our eyes are less sensitive to changes in blue, as you can see from the normalized cone response for high frequencies.

## Question 3

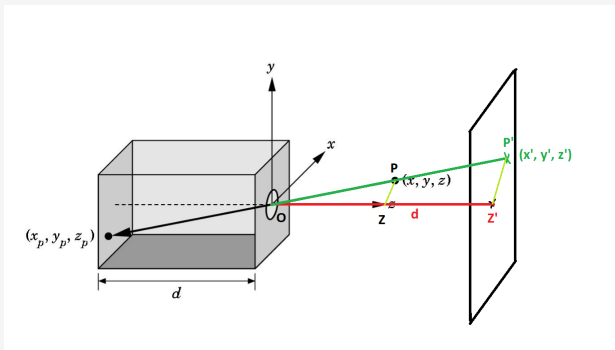
Referring to Lecture 1 Slide 31. If an imaginary image plane is  $d$  unit distance in front of the pinhole camera, what are the coordinates of the projection (on the imaginary image plane) of the 3D point  $(x, y, z)$ ?

# Similar triangles



$$\frac{AE}{AD} = \frac{AF}{AG} = \frac{EF}{DG}$$

# Perspective

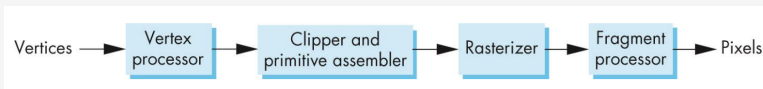


Notice that  $OPZ$  and  $OP'Z'$  are similar triangles as their internal angles are identical.

$$\frac{x}{x'} = \frac{y}{y'} = \frac{z}{z'} \text{ and by definition } z' = d$$

## Question 4

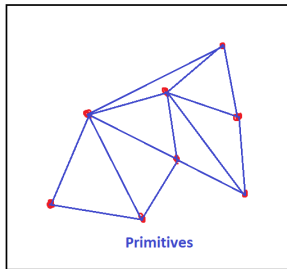
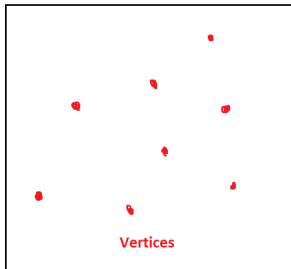
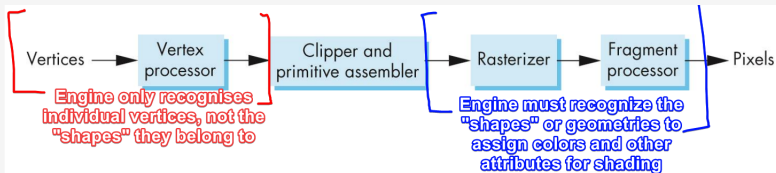
Why do we need a **primitive assembly** stage in the rendering pipeline architecture?



**Primitive:** One polygonal unit

# Primitive Assembly

## Rendering pipeline





## Question 5

What does the rasterization stage (rasterizer) do in the rendering pipeline architecture?

## Question 5

Describe what it does to a triangle that is supposed to be filled, and the three vertices have different color. Assume smooth shading is turned on.

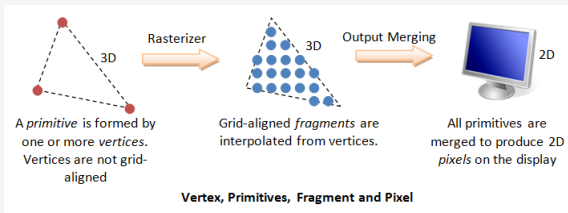
# Rasterization

## Rendering pipeline

(Lecture 1 Slide 40)

Assigning colors to pixels occupied by a primitive/polygon.

1. Each vertex has an attribute.
2. Each attribute is **interpolated** across the vertices.

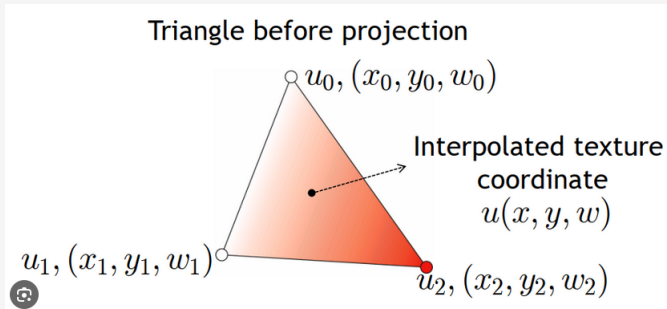


# Rasterization

Which attributes can be interpolated?

- Position
- Colour
- Normals
- UV coordinates
- Anything else that vertices may describe

We use **barycentric coordinates**.

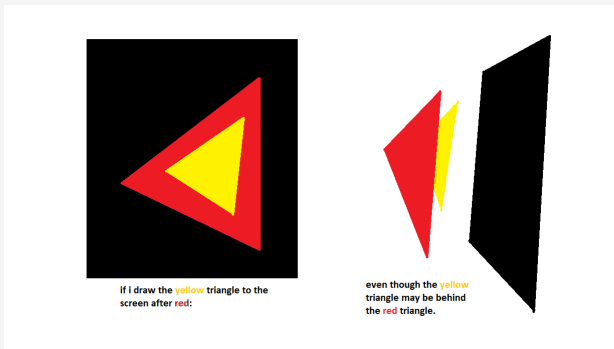


## Question 6

What is hidden-surface removal? When is it not necessary?

# Hidden Surface Removal

After rasterization: we can have multiple fragments with the same pixel coordinates.



The correct fragments to render are those that are **closest to the camera** at the same pixel coordinates.

We should discard or overwrite entire hidden surfaces/fragments.

# Hidden Surface Removal

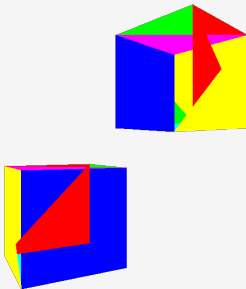
Hence if

1. we have no overlapping surfaces, or
2. we are already rendering from back to front

then we don't need hidden surface removal.

HSR also has the added benefit of reducing rendering workload (remove pixels/whole surfaces from rendering pipeline).

# Some hidden surface removal techniques





## Question 7

Which of the two following program fragments is more efficient? Why?

A	B
<pre>double v[3*N][3]; ... for ( int i = 0; i &lt; 3*N; i+=3 ) {     glBegin(GL_TRIANGLE);     glVertex3dv( v[i] );     glVertex3dv( v[i+1] );     glVertex3dv( v[i+2] );     glEnd(); }</pre>	<pre>double v[3*N][3]; ... glBegin(GL_TRIANGLE); for ( int i = 0; i &lt; 3*N; i+=3 ) {     glVertex3dv( v[i] );     glVertex3dv( v[i+1] );     glVertex3dv( v[i+2] ); } glEnd();</pre>

Can the same optimization be done for the case of GL\_POLYGON?

## Calls to glBegin and glEnd

A	B
<pre>double v[3*N][3]; ... for ( int i = 0; i &lt; 3*N; i+=3 ) {     glBegin(GL_TRIANGLE);     glVertex3dv( v[i] );     glVertex3dv( v[i+1] );     glVertex3dv( v[i+2] );     glEnd(); }</pre> <p><i>N times</i></p>	<pre>double v[3*N][3]; ... glBegin(GL_TRIANGLE); for ( int i = 0; i &lt; 3*N; i+=3 ) {     glVertex3dv( v[i] );     glVertex3dv( v[i+1] );     glVertex3dv( v[i+2] ); } glEnd();</pre> <p><i>once!</i></p>

Note that OpenGL (together with mosts other graphics engines) is a **state machine**. Method B greatly reduces the number of state changes.

# What about GL\_POLYGON?

We can't do this with GL\_POLYGON or we'll be defining one massive  $3N$ -vertex polygon.

GL\_POLYGON

Draws a single, convex polygon. Vertices 1 through  $N$  define this polygon.

# WebGL State Machine visualizer/debugger tool

The image displays the WebGL State Machine visualizer/debugger tool interface. It features several interconnected panels:

- global state**: A panel on the left showing various WebGL state variables. The **Texture Units** section is expanded, showing a table of texture units (0-7) with their current texture (CUBE\_MAP or null). The **clear state** section shows **DEPTH\_TEST** as false, **DEPTH\_FUNC** as LESS, **DEPTH\_RANGE** as 0, 1, and **DEPTH\_WRITEMASK** as true.
- program[prg]**: A panel showing the attached shaders and attribute info. The **uniforms** section is expanded, showing the **state** of the program, including **LINK\_STATUS** as true.
- canvas**: A panel showing the canvas element, which is currently displaying a white square.
- buffer[positionBuffer]**: A panel showing the data for the position buffer, which is a Float32Array containing the values [0, 0.7, 0.5, -0.7, -0.5, -0.7].
- buffer[colorBuffer]**: A panel showing the data for the color buffer, which is a Uint8Array containing the values [255, 0, 0, 255, 0, 255, 0, 255].
- vertex array[default\*]**: A panel showing the attributes of the vertex array. The **attributes** section is expanded, showing a table of attributes (0-7) with their enabled status, size, type, normalized status, stride, offset, divisor, and buffer. The **state** section shows **ELEMENT\_ARRAY\_BUFFER\_BINDING** as null.
- code editor**: A panel on the right showing the WebGL code being executed. The code is for a "Rainbow Triangle" and includes the following snippets:

```
const vertexPositions = new Float32Array([
  0, 0.7,
  0.5, -0.7,
  -0.5, -0.7,
]);

const positionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
gl.bufferData(gl.ARRAY_BUFFER, vertexPositions, gl.STATIC_DRAW);

const vertexColors = new Uint8Array([
  255, 0, 0, 255,
  0, 255, 0, 255,
  0, 0, 255, 255,
]);

const colorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
gl.bufferData(gl.ARRAY_BUFFER, vertexColors, gl.STATIC_DRAW);

gl.enableVertexAttribArray(positionLoc);
gl.vertexAttribPointer(
  positionLoc,
  2, // 2 values per vertex shader iteration
  gl.FLOAT, // data is 32bit floats
  false, // don't normalize
  0, // stride (0 = auto)
  0, // offset into buffer
);

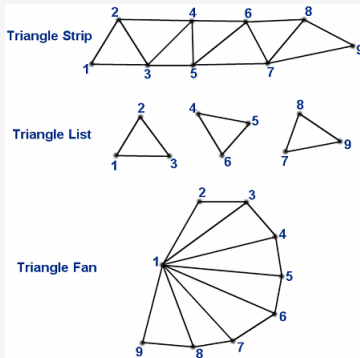
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
gl.enableVertexAttribArray(colorLoc);
gl.vertexAttribPointer(
  colorLoc,
  4, // 4 values per vertex shader iteration
  gl.UNSIGNED_BYTE, // data is 8bit unsigned bytes
  true, // do normalize
  0, // stride (0 = auto)
  0, // offset into buffer
);

gl.useProgram(prg);

// compute 3 vertices for 1 triangle
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

## Question 8

OpenGL supports the `GL_TRIANGLES` primitive type. Why do you think that OpenGL also supports `GL_TRIANGLE_FAN` and `GL_TRIANGLE_STRIP`?



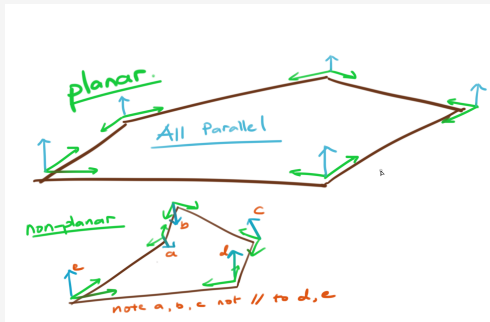
# Comparison

Type	Vertices	Triangles
GL_TRIANGLES	$3n$	$n$
GL_TRIANGLE_FAN	$n + 2$	$n$
GL_TRIANGLE_STRIP	$n + 2$	$n$

## Question 9

Devise a test to check whether a polygon in 3D space is planar.

## Method from Tutorial answer



1. For each vertex, take the cross product of its two neighbouring edges.
2. Normalize the cross product.
3. Identify if all the cross products of each vertex are the same.

Slow because cross product computation is expensive.

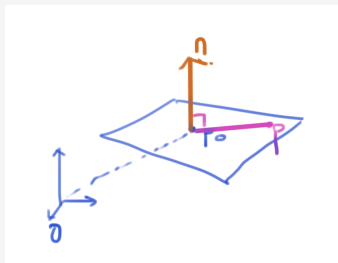


# Improved method

Note that a plane can be defined by the following equation:

$$n \cdot (p - p_0) = 0$$

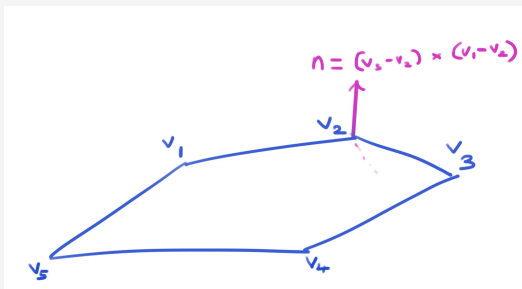
where  $n$  is the **normal of the plane**.



This is because dot product of two vectors is zero if and only if they are perpendicular!

# Improved method

Hence let's set the normal of this polygon to be  $(v_1 - v_2) \times (v_3 - v_2)$ .



# Improved Method

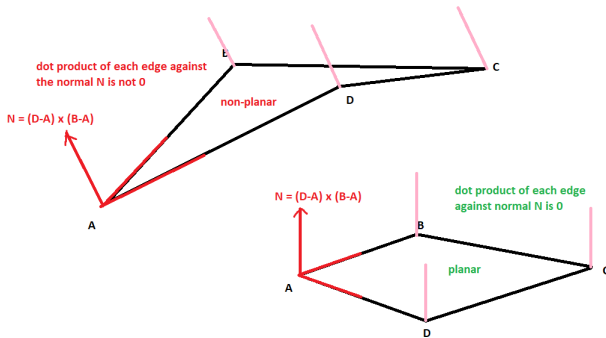
For each of the vertices  $v_i \in \{v_1, v_2, v_3, \dots, v_n\}$ , if

$$n \cdot (v_i - v_2) = 0$$

then the polygon is planar.

Cross product involves more operations than dot product, so dot product is more efficient to compute. Asymptotically the runtime is still the same ( $O(n)$ ).

# Summary

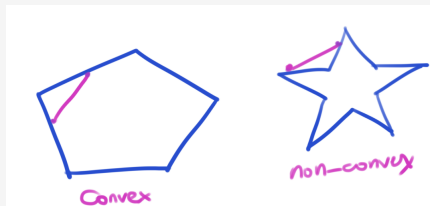


## Question 10

Devise a test to check whether a polygon on the x-y plane is convex.

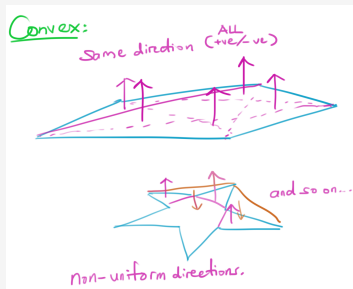
# What is a convex polygon

Convex polygon is a polygon that is identical to its **convex hull**.



If you choose any two points on the boundary of a convex polygon and draw a line between, the line should be entirely contained within the polygon.

# Method from Tutorial answer



Iterating through  $i = 0$  to  $n - 1$  where there are  $n$  vertices, find each

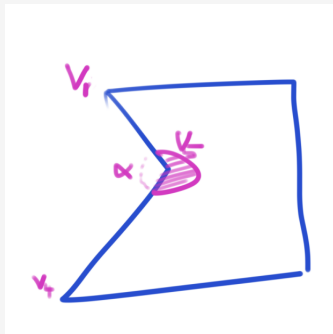
$$n_i = \|(v_{i-1} - v_i) \times (v_{i+1} - v_i)\|$$

if **at least one**  $n_i$  **is not identical to the rest**, then polygon is non-convex.

Note: I'm omitting that you have to take the  $x \bmod n$  of each of the  $v_x$  here for conciseness.

## What about inner angles?

One property of convex polygons: all interior angles  $< 180^\circ$



Checking this is tricky as the dot product of the vectors  $v_4 - v_3$  and  $v_2 - v_3$  gives you  $\cos \alpha$ , not the cosine of the interior angle  $2\pi - \alpha$ . One way would be to compute the cross product, which we would have done so using the other method anyway.



Thanks! Get the slides here.



<https://trxe.github.io/cs3241-notes>