

CS4533 Lecture 6

Slides/Notes

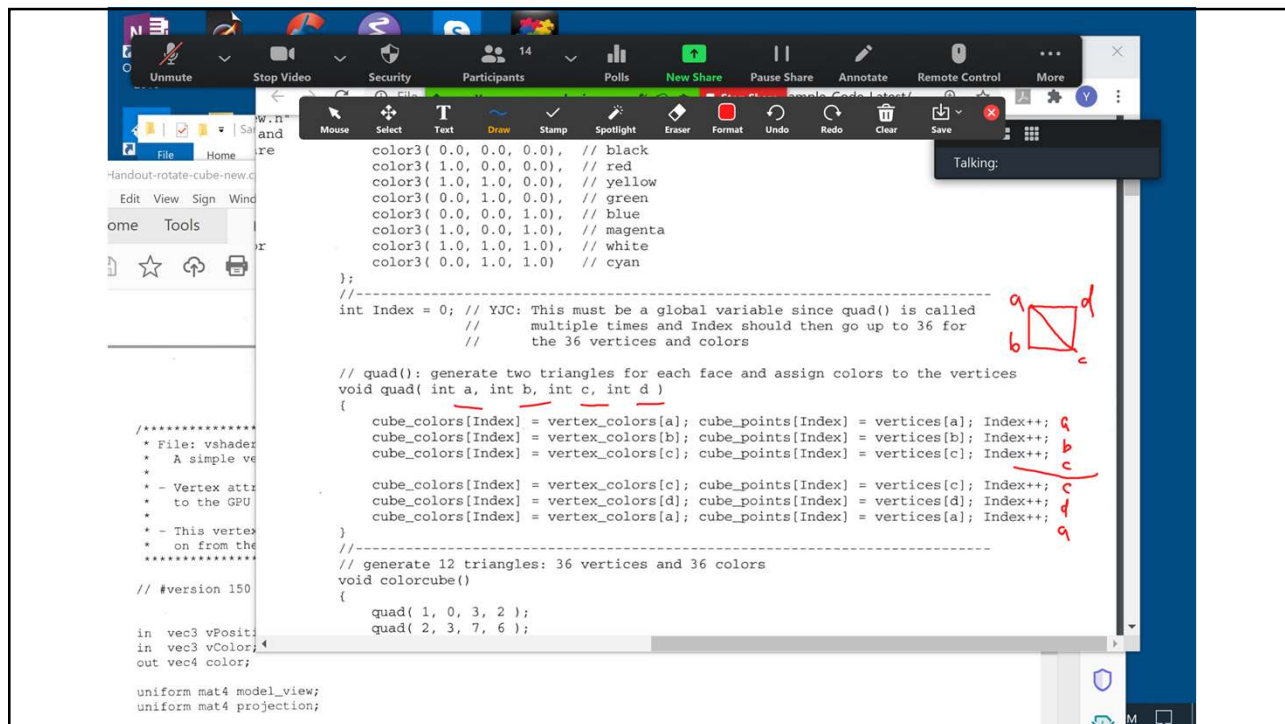
**Sample Code for Shader-Based OpenGL,
Sphere Rolling Transformations in HW2,
Polygon Scan Conversion
(Handouts, Notes)**

By Prof. Yi-Jen Chiang
CSE Dept., Tandon School of Engineering
New York University

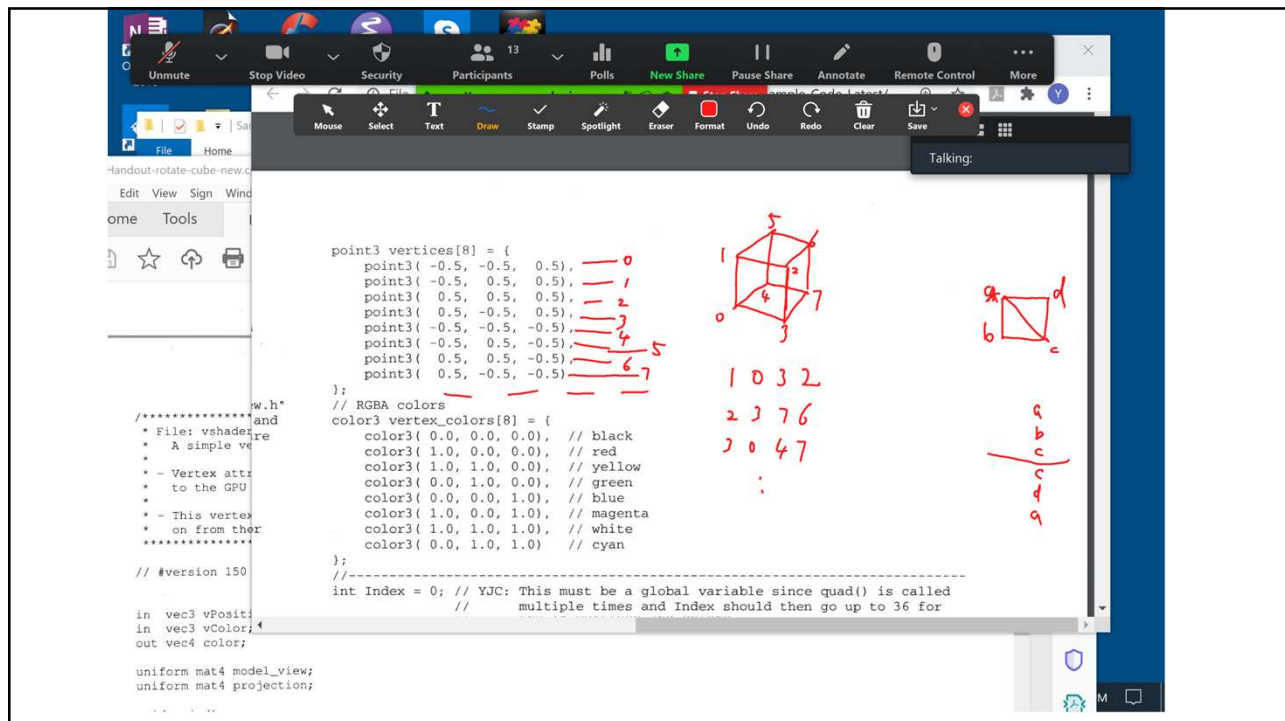
1

- Discussed the sample code ``Handout: rotoate-cube-new.cpp'' (posted at ``<https://cse.engineering.nyu.edu/cs653/>'' under '[Handout: rotate-cube-new.cpp \("Rotate-Cube-New.tar.gz"\)](#)'). You can download the source code and play with it. **You can use it as the starting point of HW2.**
- Some screen shots of the sample code discussions with annotations are listed below.

2



3



4

```

// the 36 vertices and colors

// quad(): generate two triangles for each face and assign colors to the vertices
void quad( int a, int b, int c, int d )
{
    cube_colors[Index] = vertex_colors[a]; cube_points[Index] = vertices[a]; Index++;
    cube_colors[Index] = vertex_colors[b]; cube_points[Index] = vertices[b]; Index++;
    cube_colors[Index] = vertex_colors[c]; cube_points[Index] = vertices[c]; Index++;

    cube_colors[Index] = vertex_colors[c]; cube_points[Index] = vertices[c]; Index++;
    cube_colors[Index] = vertex_colors[d]; cube_points[Index] = vertices[d]; Index++;
    cube_colors[Index] = vertex_colors[a]; cube_points[Index] = vertices[a]; Index++;
}

// generate 12 triangles: 36 vertices and 36 colors
void colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}

// generate 2 triangles: 6 vertices and 6 colors
void floor()
{
    floor_colors[0] = vertex_colors[3]; floor_points[0] = vertices[3];
    floor_colors[1] = vertex_colors[0]; floor_points[1] = vertices[0];
    floor_colors[2] = vertex_colors[4]; floor_points[2] = vertices[4];
    floor_colors[3] = vertex_colors[4]; floor_points[3] = vertices[4];
}

```

Handwritten notes on slide 5 include a cube diagram with vertices 0-7, a square diagram with vertices a, b, c, d, and a vertical list of colors: a, b, c, d, e, f.

5

```

// generate 12 triangles: 36 vertices and 36 colors
void colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}

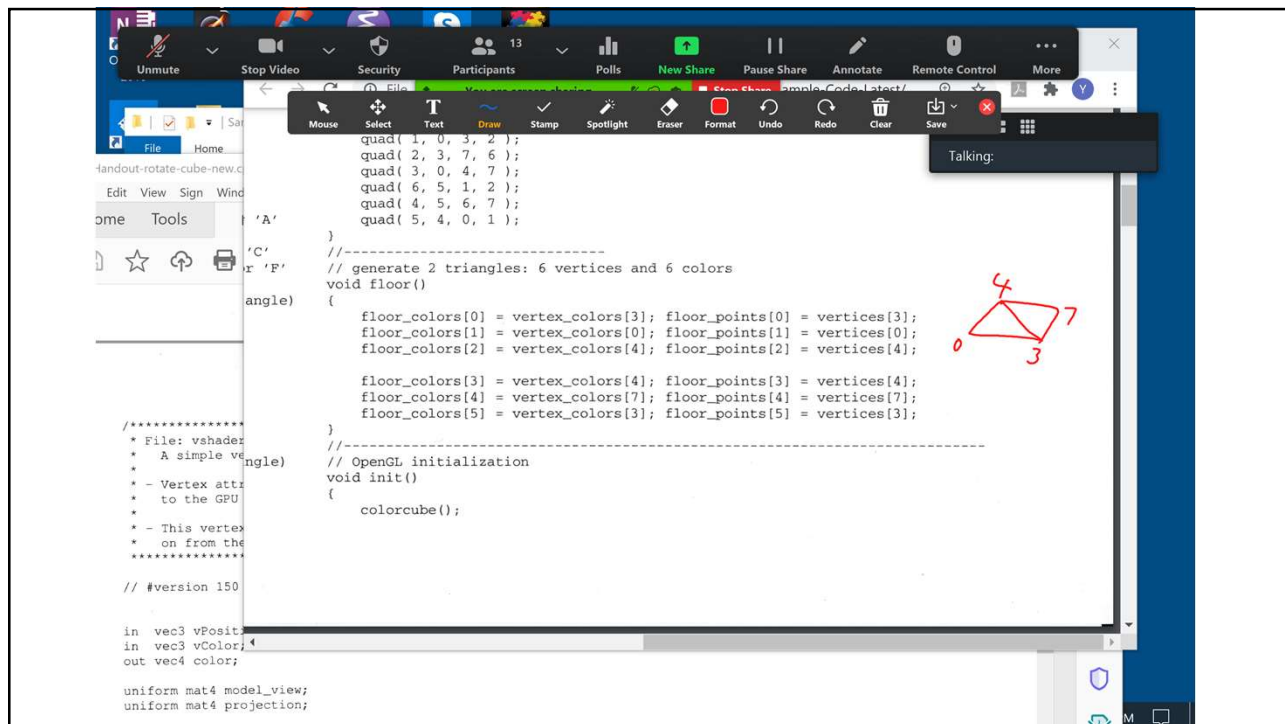
// generate 2 triangles: 6 vertices and 6 colors
void floor()
{
    floor_colors[0] = vertex_colors[3]; floor_points[0] = vertices[3];
    floor_colors[1] = vertex_colors[0]; floor_points[1] = vertices[0];
    floor_colors[2] = vertex_colors[4]; floor_points[2] = vertices[4];
    floor_colors[3] = vertex_colors[4]; floor_points[3] = vertices[4];
    floor_colors[4] = vertex_colors[7]; floor_points[4] = vertices[7];
    floor_colors[5] = vertex_colors[3]; floor_points[5] = vertices[3];
}

// OpenGL initialization
void init()
{
    colorcube();
}

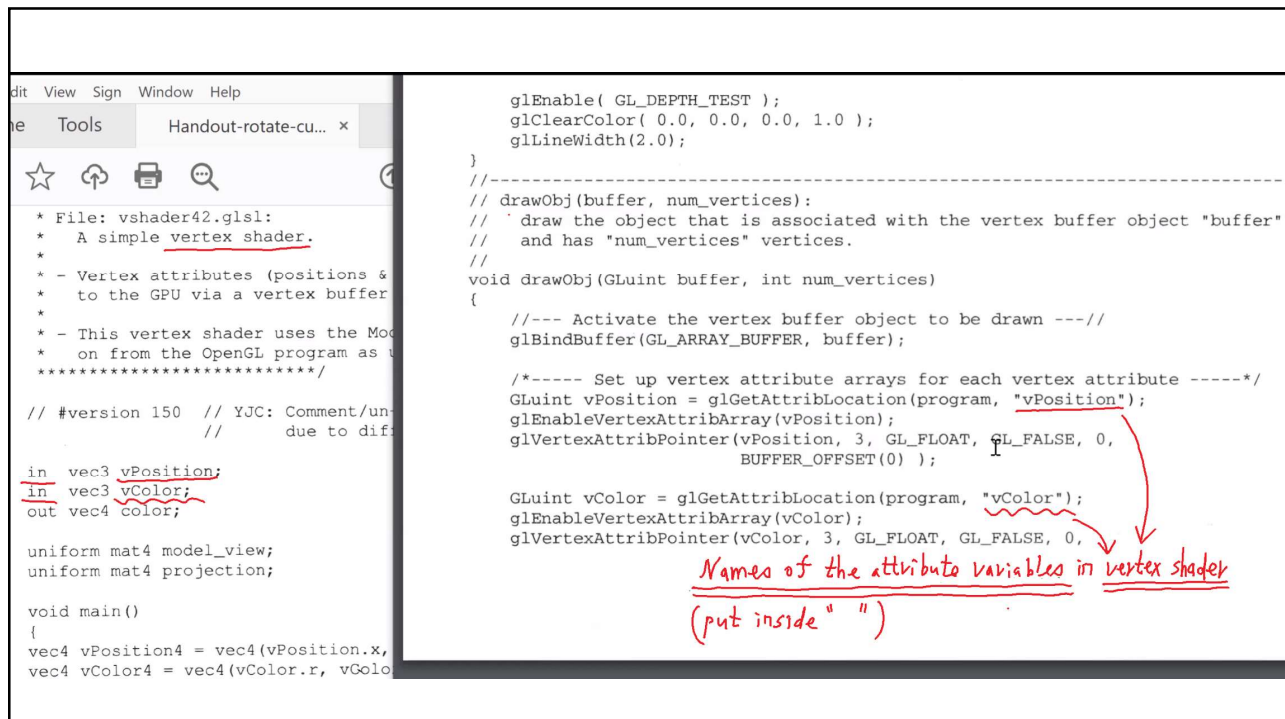
```

Handwritten notes on slide 6 include a cube diagram with vertices 0-7, a square diagram with vertices a, b, c, d, and a vertical list of colors: a, b, c, d, e, f.

6



7



8

```

program = InitShader("vshader42.glsl", "fshader42.glsl");

glEnable( GL_DEPTH_TEST );
glClearColor( 0.0, 0.0, 0.0, 1.0 );
glLineWidth(2.0);

//-----
/ drawObj(buffer, num_vertices):
/ draw the object that is associated with the vertex buffer object "buffer"
/ and has "num_vertices" vertices.
/
void drawObj(GLuint buffer, int num_vertices)


    //--- Activate the vertex buffer object to be drawn ---//
    glBindBuffer(GL_ARRAY_BUFFER, buffer);

    /*----- Set up vertex attribute arrays for each vertex attribute -----*/
    GLuint vPosition = glGetAttribLocation(program, "vPosition");
    glEnableVertexAttribArray(vPosition);
    glVertexAttribPointer(vPosition, 3, GL_FLOAT, GL_FALSE, 0,
        BUFFER_OFFSET(0) );

    GLuint vColor = glGetAttribLocation(program, "vColor");
    glEnableVertexAttribArray(vColor);
    glVertexAttribPointer(vColor, 3, GL_FLOAT, GL_FALSE, 0,
        color

    ↑ ↑: 1st vertex
    ↑ ↑: 2nd vertex

```



9

```

{
    colorcube();

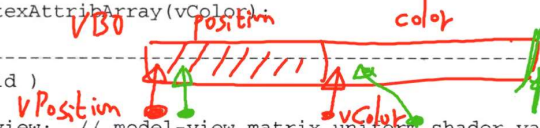
    BUFFER_OFFSET(sizeof(point3) * num_vertices) );
    // the offset is the (total) size of the previous vertex attribute array(s)

    /* Draw a sequence of geometric objs (triangles) from the vertex buffer
       (using the attributes specified in each enabled vertex attribute array) */
    glDrawArrays(GL_TRIANGLES, 0, num_vertices);

ay()
    /*--- Disable each vertex attribute array being enabled ---*/
    glDisableVertexAttribArray(vPosition);
    glDisableVertexAttribArray(vColor);
}

//-----
void display( void )
{
    GLuint model_view; // model-view matrix uniform shader variable location
    GLuint projection; // projection matrix uniform shader variable location

```



10


```

/*----- Set Up the Model-View matrix for the cube -----*/
#if 0 // The following is to verify the correctness of the function NormalMatrix():
    // Commenting out Rotate() and un-commenting mat4WithUpperLeftMat3()
    // gives the same result.
    mv = mv * Translate(0.0, 0.5, 0.0) * Scale (1.4, 1.4, 1.4)
        * Rotate(angle, 0.0, 0.0, 2.0);
    // * mat4WithUpperLeftMat3(NormalMatrix(Rotate(angle, 0.0, 0.0, 2.0), 1));
#endif
#if 1 // The following is to verify that Rotate() about (0,2,0) is RotateY():
    // Commenting out Rotate() and un-commenting RotateY()
    // gives the same result.
    //
    // The set-up below gives a new scene (scene 2), using Correct LookAt().
    mv = mv * Translate(0.0, 0.5, 0.0) * Scale (1.4, 1.4, 1.4)
        * Rotate(angle, 0.0, 2.0, 0.0);
    // * RotateY(angle);
    //
    // The set-up below gives the original scene (scene 1), using Correct LookAt().
    mv = Translate(0.0, 0.5, 0.0) * mv * Scale (1.4, 1.4, 1.4)
        * Rotate(angle, 0.0, 2.0, 0.0);
    // * RotateY(angle);
    //
    // The set-up below gives the original scene (scene 1), when using previously
    // Incorrect LookAt() (= Translate(1.0, 1.0, 0.0) * correct LookAt() )
    mv = Translate(-1.0, -0.5, 0.0) * mv * Scale (1.4, 1.4, 1.4)
        * Rotate(angle, 0.0, 2.0, 0.0);
    // * RotateY(angle);
    //
#endif
#if 0 // The following is to verify that Rotate() about (3,0,0) is RotateX():
    // Commenting out Rotate() and un-commenting RotateX()

```

model-view matrix
 mv is set as
 $mv \leftarrow \text{LookAt}() * \text{Scale}() * \text{Rotate}()$
 ① First apply Rotate() to objects
 ② Then apply Scale() to obj.
 These are all in the world frame
 ③ Finally apply LookAt() to change to eye frame

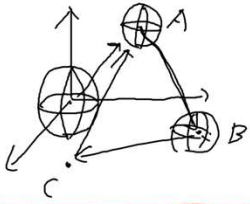
11

Transformations for Sphere Rolling in HW2

- Discussed how to do transformations for **sphere rolling** --- there are two transformations: **rotation** & **translation**. They have to **match each other** (**rotation amount** corresponds to the **distance of translation**) to produce the sphere rolling effect.
- Discussed **HW2 part (c)**: Treat each rotation segment (sphere center going from A to B, from B to C, from C to A) **independently** (see next slides).
- HW2 part (d) (correct rolling transition)**: build on top of part (c) (see the slide "HW2 Part (d)" below).

12

HW2: Part (c) Sphere Rolling.



Initial S.O.L.: Treat each rolling segment independently.

1. Rotate (angle, V_x, V_y, V_z)
(in degrees) Rotation axis vector

$$\vec{AB} = B - A. \quad \vec{OY} = (0, 1, 0)$$

Suppose $\vec{AB} = (1, 2, 3)$ \times $\vec{AB} = (1, 2, 3)$

$$\vec{OY} \times \vec{AB} = (3, 0, -1) = (V_x, V_y, V_z)$$

($\vec{AB} \times \vec{OY}$ is wrong: rotating backwards)

* $\vec{OY} = (0, 1, 0)$: normal vector to the ground.
 $\vec{OY} \perp$ ground.

* Rotation axis vector (V_x, V_y, V_z) is \perp to both \vec{OY} & \vec{AB}

$$\Rightarrow (V_x, V_y, V_z) = \vec{OY} \times \vec{AB}.$$

13

(Input sphere is centered at the origin)

2. Translate (T_x, T_y, T_z)

$$(T_x, T_y, T_z) = \vec{OA} + d \cdot \frac{\vec{AB}}{|\vec{AB}|}$$

length 1

* Applying to the sphere:

- 1 Rotate ()
 - 2 Translate ()
 - 3 LookAt ()
- in this order

eg. $\vec{AB} = (1, 2, 3)$ $\frac{\vec{AB}}{|\vec{AB}|} = \frac{(1, 2, 3)}{\sqrt{1^2 + 2^2 + 3^2}} = \left(\frac{1}{\sqrt{14}}, \frac{2}{\sqrt{14}}, \frac{3}{\sqrt{14}}\right)$

$$\frac{360}{2\pi r} = \frac{\text{angle}}{d} \Rightarrow d = \frac{\text{angle}}{360} \cdot 2\pi r$$

(r=1)

* angle:

- * Global variable, initially 0
- * In each frame, increase its value by some fixed amount

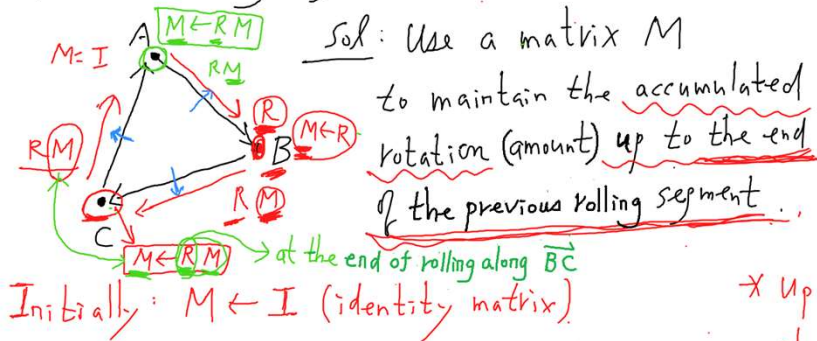
Note: 1. Use the variable "angle" to control both Rotate() and Translate() so that they match each other to produce the sphere rolling effect.

2. When angle > 360. do we reset angle $\leftarrow 0$? Ans: No (angle $\leftarrow 0 \Rightarrow d = 0$)
(This puts the sphere back to center at A.)

14

HW2 Part (d): Fix the issue of

incorrect Rolling Segment Transition.



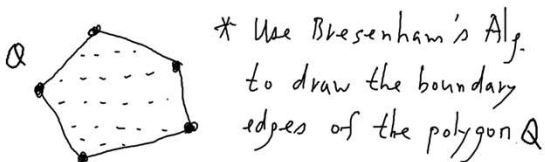
× Update M each time when changing the rolling segment

- ① M
 - ② Rotate()
 - ③ Translate()
 - ④ LookAt()
- Rolling as in part (c)
independently along the current seg.

15

Scan-Conversion of Polygons

Purpose: Display filled polygons.



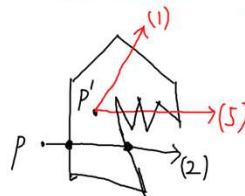
First Issue: Given the boundary.

decide the interior of the polygon Q .

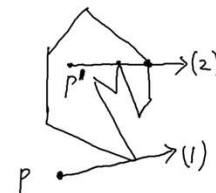
- (1) Odd-Even test
 - (2) Winding test.
- For a pt p decide whether p is inside or outside Q

(1) Odd-Even test:

Draw a ray from p to infinity.
count # intersections between the ray and the boundary: odd: p is inside
even: p is outside

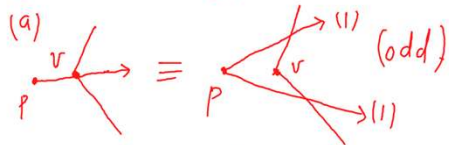


Issue: What if the ray goes thru a vertex?
Degenerate cases!
Singularity

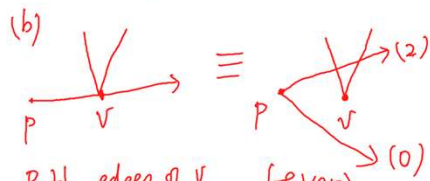


16

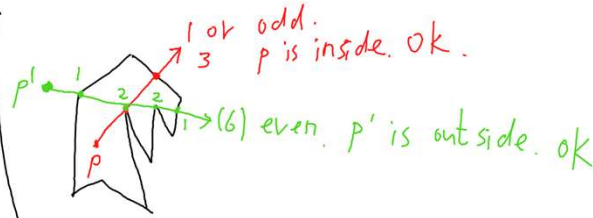
Solution: The perturbation idea:



For vertex v where its 2 edges are on both sides of the ray \Rightarrow 1 intersection.

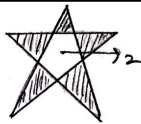


Both edges of v are on the same side of the ray \Rightarrow 0 or 2 intersections (-even)



(2) Winding Test (see next)

17



How about



(2) Winding test



point P

① any ray \vec{PQ} to infinity, not going thru any vertex

② d' : rotate \vec{PQ} counter clock wise by 90°

③ Trace all edges of the polygon by following the edges to enclose the polygon once \Rightarrow each edge is assigned a direction.

④ For each edge e crossed by \vec{PQ} , project e along d' : same as d' : +1
opposite to d' : -1

sum $\neq 0$: P is inside
= 0: outside

eg: e_1 : -1

e_2 : -1

sum = -2 $\therefore P$ is inside

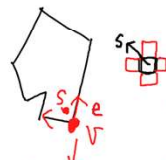
#

18

Next: How do we fill the interior of a polygon?

M1: Flood Fill: ① Pick a seed pt S inside the polygon

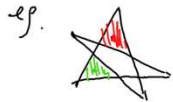
② grow until hitting the boundary.



* Finding a seed S:
Lowest vertex v go along the right edge e of v
Pick a pt to the left of e .

19

* If the polygon boundary crosses itself, then we need a seed in each region.



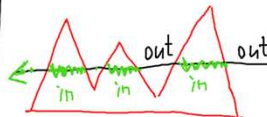
For a simple polygon (boundary never crosses) a single seed is enough.

M2: Scan-line Algorithm.

Def: Scan line: A horizontal line of pixels that goes thru the frame buffer end to end



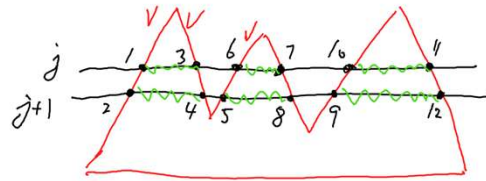
* Use odd-even test



20

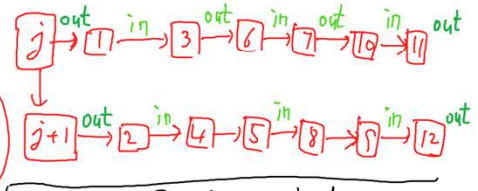
* For each scan line, maintain a list of intersections with the polygon boundary sorted left-to-right:

⇒ Use x - x algorithm:



- ① Create a bucket for each scan line.
- ② Process each edge, put the intersections with the scan lines to the corresponding buckets. (when drawing each pt on the edge, its y -value indicates the scan line intersected)

- ③ Each pt is inserted to the bucket by insertion sort according to the x -value. (left → right)



Q: In step ②, if we instead process each scan line and try to find its intersections with edges, ?

Ans: Correct, but inefficient.

Need to check each edge, but may find no intersection. (wasted ops)