COMP3421

Inverse Transformations, Shaders

Robert Clifton-Everest

Email: robertce@cse.unsw.edu.au

View transform

- The world is rendered as it appears in the camera's local coordinate frame.
- The view transform converts the world coordinate frame into the camera's local coordinate frame.
- Note that this is the inverse of the transformation that would convert the camera’s local coordinate frame into world coordinates.

Transformation pipeline

- We transform in 2 stages

$$P_{camera} \xleftarrow{view} P_{world} \xleftarrow{model} P_{local}$$

- The model transform transforms points in the local coordinate system to the world coordinate system

- The view transform transforms points in the world coordinate system to the camera's coordinate system

View transform

- Mathematically if:

$$P_{world} = Trans(Rot(Scale(P_{camera})))$$

- Then the view transform is:

$$P_{camera} = Scale^{-1}(Rot^{-1}(Trans^{-1}(P_{world})))$$

Inverse Transformations

- If the local-to-global transformation is:

$$Q = \mathbf{M_T}\mathbf{M_R}\mathbf{M_S}P$$

- then the global-to-local transformation is the inverse:

$$P = \mathbf{M_S^{-1}}\mathbf{M_R^{-1}}\mathbf{M_T^{-1}}Q$$

Inverse Transformations

- Inverses are easy to compute:

$$\begin{array}{rrcl}
\text{translation:} & \mathbf{M_T}^{-1}(d_x, d_y) & = & \mathbf{M_T}(-d_x, -d_y) \\
\text{rotation:} & \mathbf{M_R}^{-1}(\theta) & = & \mathbf{M_R}(-\theta) \\
\text{scale:} & \mathbf{M_S}^{-1}(s_x, s_y) & = & \mathbf{M_S}(1/s_x, 1/s_y) \\
\text{shear:} & \mathbf{M_H}^{-1}(h) & = & \mathbf{M_H}(-h)
\end{array}$$

Local to World Exercise

- Given this local coordinate frame:

```
CoordFrame2D.identity()
    .translate(3,2)
    .rotate(-45)
    .scale(0.5,0.5);
```

- What point in the local co-ordinate frame would correspond to the world co-ordinate Q (2,-1)?

Assignment 1

- Automarking
  - Junit 4 Unit Tests
  - diff image files that you output with required image output
- Tutor subjective marking
  - MyCoolSceneObject
  - Bonus Game (also course vote)

JUnit

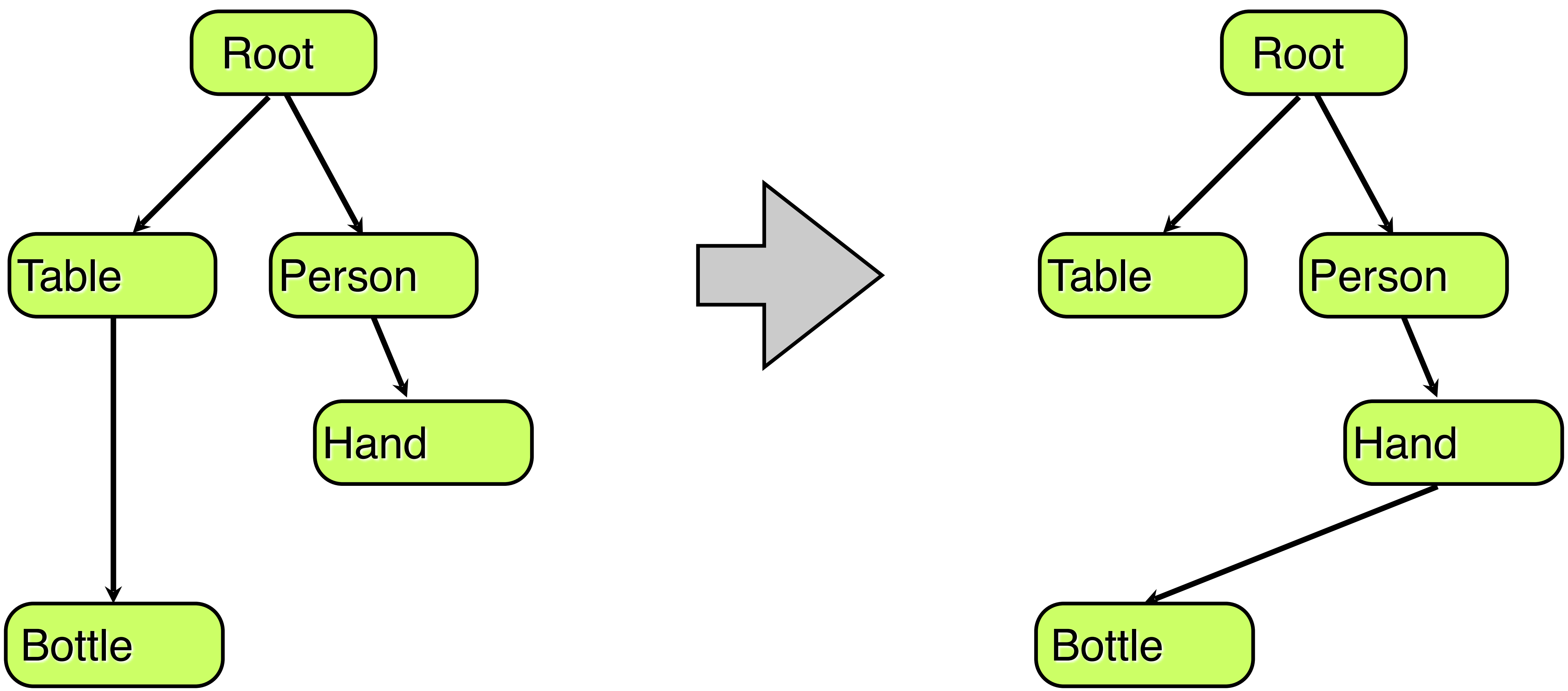
- You can run JUnit tests directly from eclipse
- For floating point equality we use an epsilon
- Demo

Hints

- Drawing twice: once as a fill and once as an outline.
  - Draw fill first
  - Then outline over the top

Reparenting

- What if the person picks up the bottle?
- What's the new transformation?



Lerping

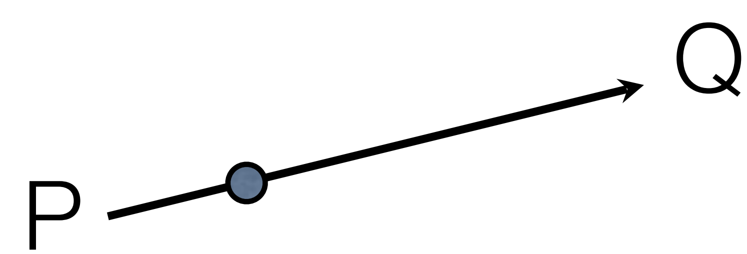
- We can add affine combinations of points:

$$\frac{1}{2}(p_1, p_2, 1)^\top + \frac{1}{2}(q_1, q_2, 1)^\top = (\frac{p_1 + q_1}{2}, \frac{p_2 + q_2}{2}, \mathbf{1})^\top$$

- We often use this to do linear interpolation between points:

$$lerp(P, Q, t) = P + t(Q - P)$$

$$lerp(P, Q, t) = P(1 - t) + tQ$$



$lerp(P, Q, 0.3)$

Lerping Exercise

- Using linear interpolation, what is the midpoint between A=(4,9) and B=(3,7)?

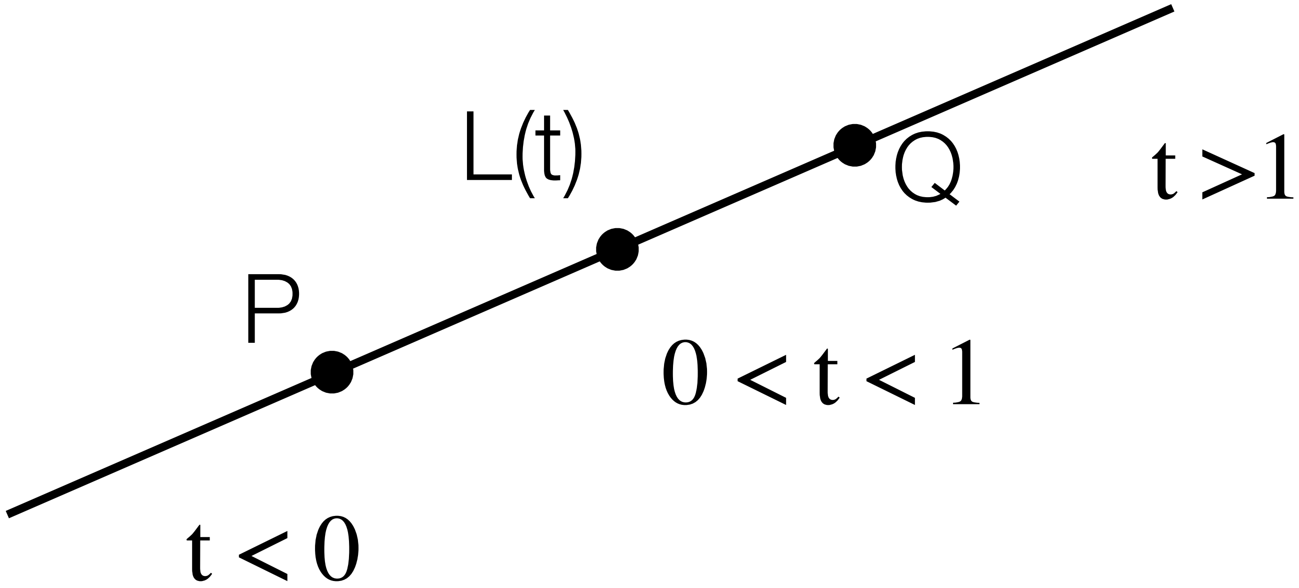
Lines

- Parametric form:

$$L(t) = P + t\mathbf{v}$$

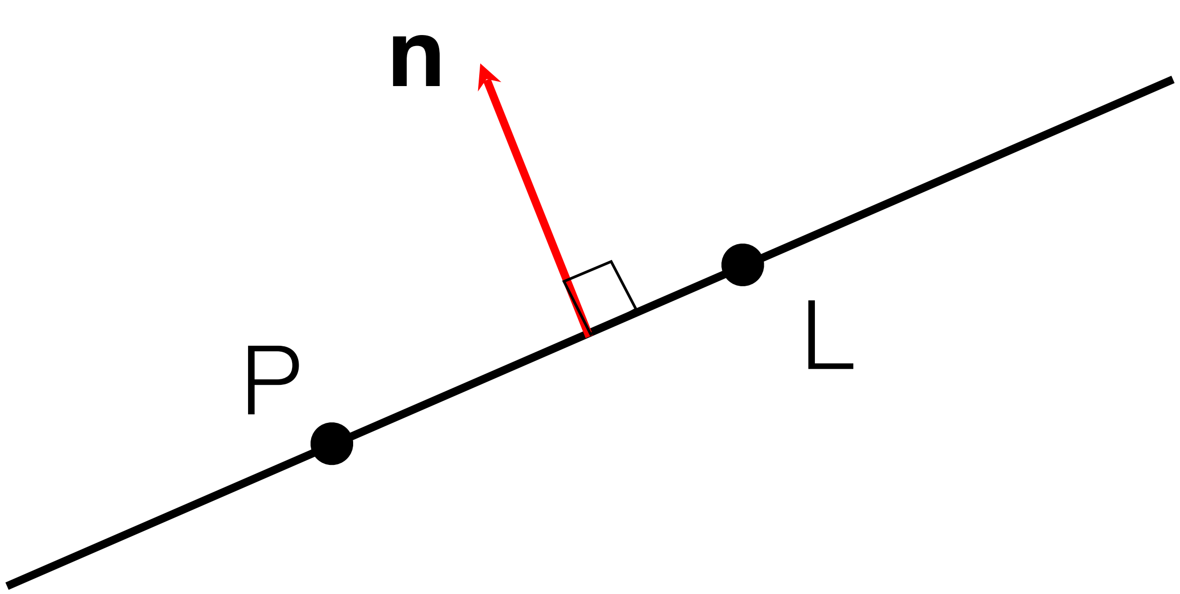
$$\mathbf{v} = Q - P$$

$$L(t) = P + t(Q - P)$$



- Point-normal form in 2D:

$$\mathbf{n} \cdot (P - L) = 0$$



Line intersection

- Two lines

$$L_{AB}(t) = A + (B - A)t$$
$$L_{CD}(u) = C + (D - C)u$$

- Solve simultaneous equations:

$$(B - A)t = (C - A) + (D - C)u$$

Line Intersection Example

A = (0,3) B = (12,7)

C = (2,0) D = (7,20)

$$L_{AB}(t) = A + (B - A)t$$
$$L_{CD}(u) = C + (D - C)u$$

Line Intersection Example 2

Find where the line $L(t) = A + \mathbf{c}t$ intersects with the line $\mathbf{n}.(P-B) = 0$ where

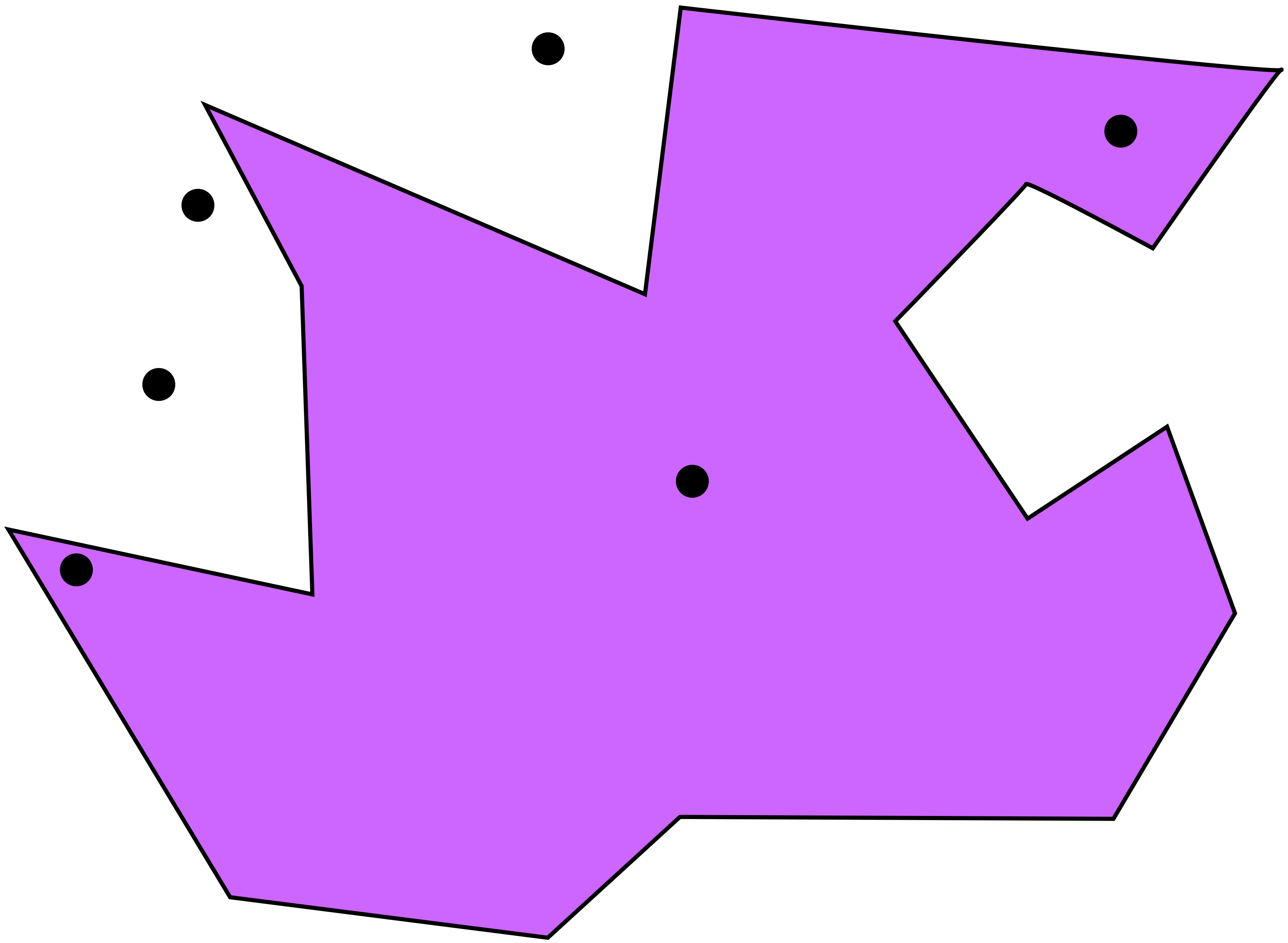
$A=(2,3)$, $\mathbf{c} = (4,-4)$, $\mathbf{n} = (6,8)$, $B=(7,7)$

NOT DONE IN LECTURE DUE TO TIME.
Left as an exercise for the reader.

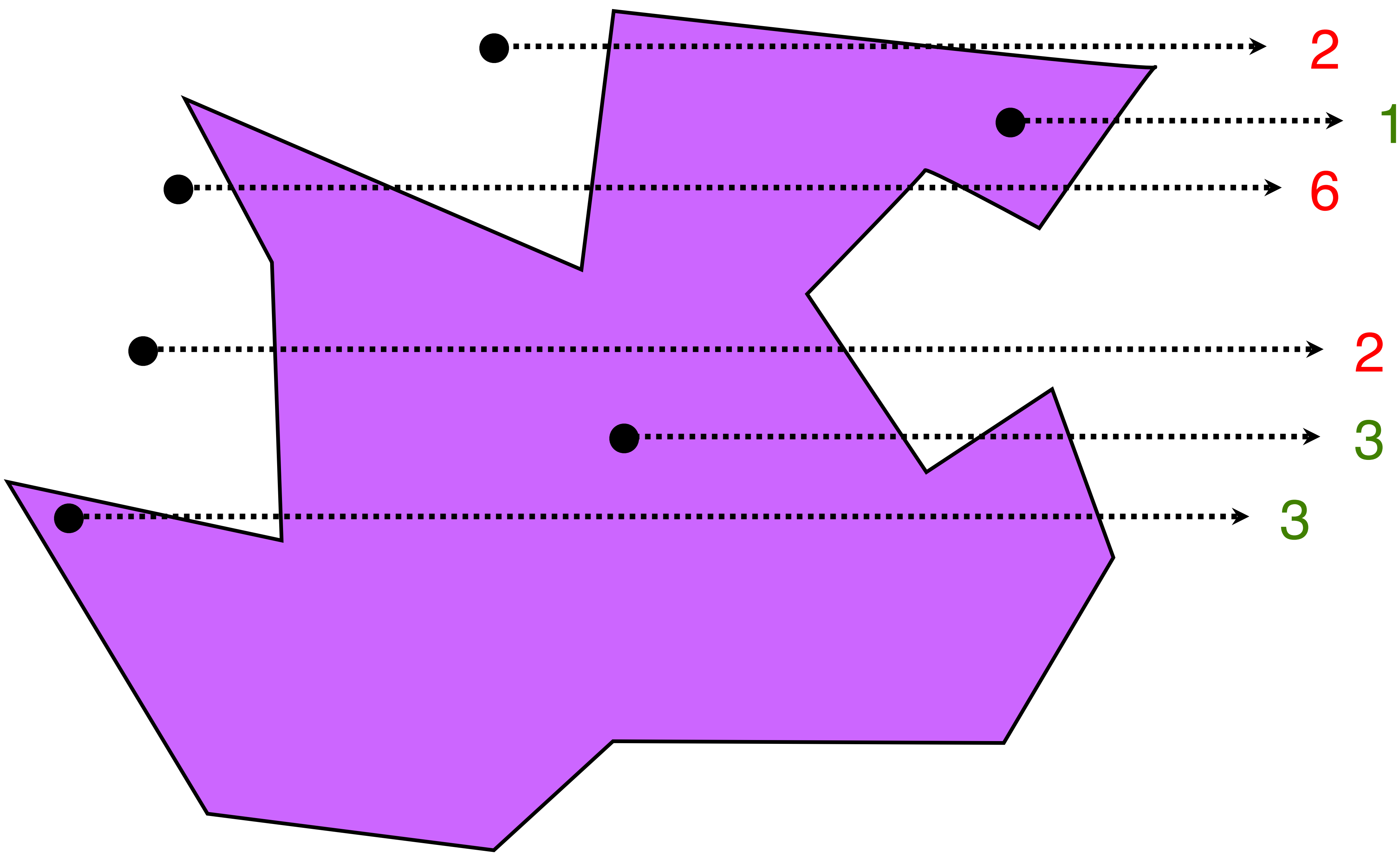
Point in Polygon

- For any ray from the point
- Count the number of crossings with the polygon
- If there is an odd number of crossings the point is inside

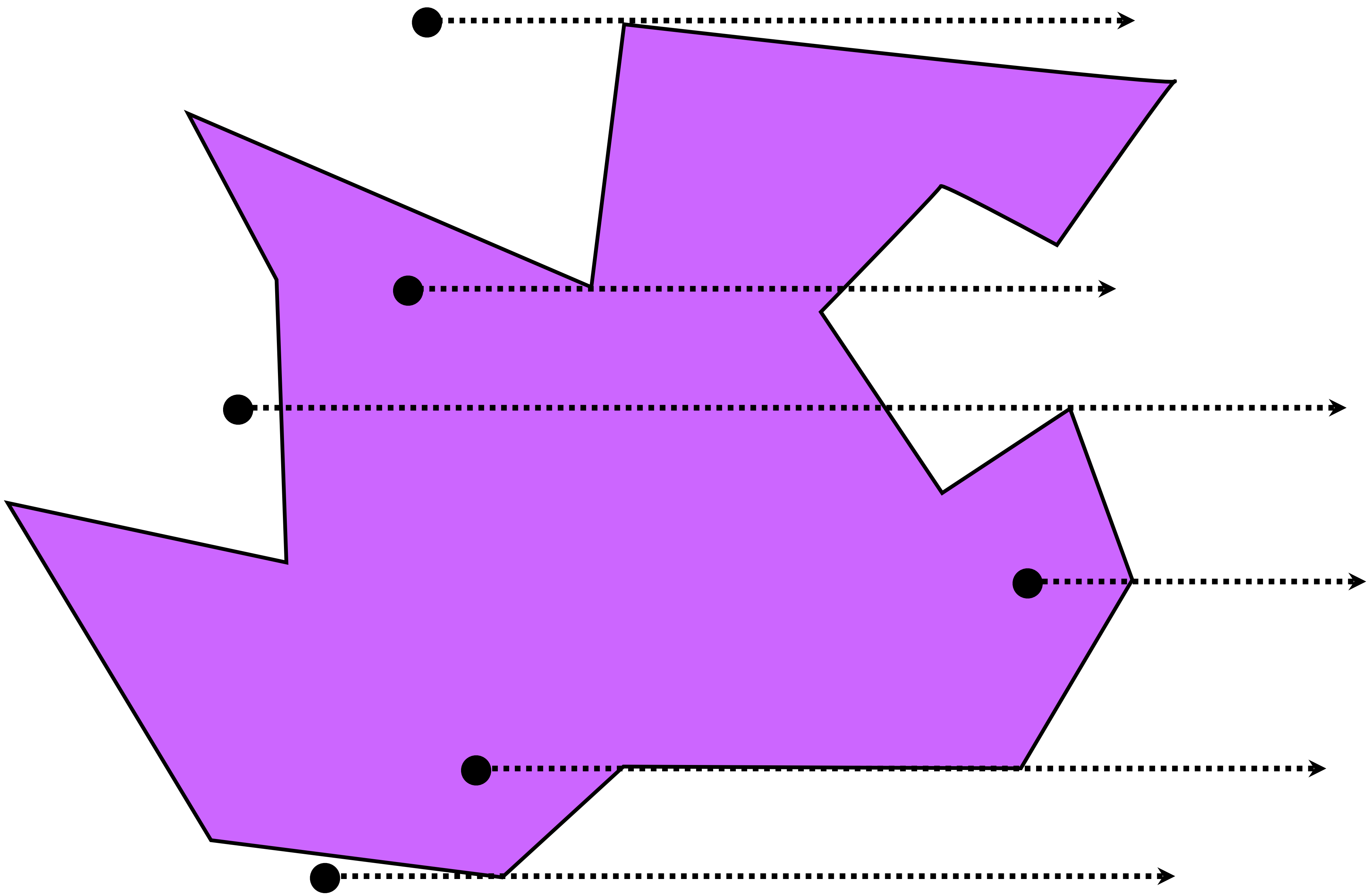
Point in polygon



Point in polygon

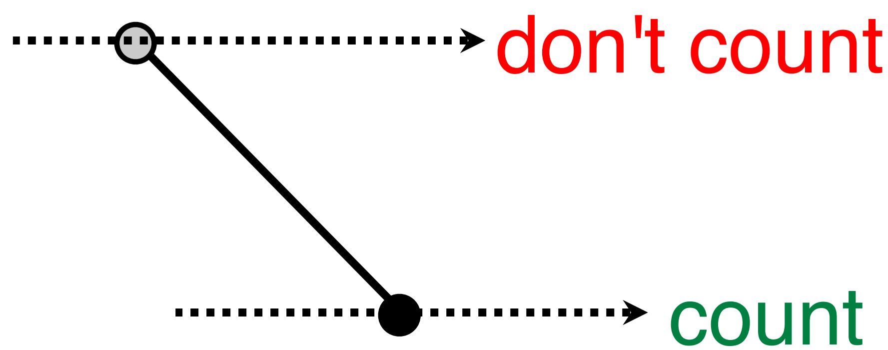


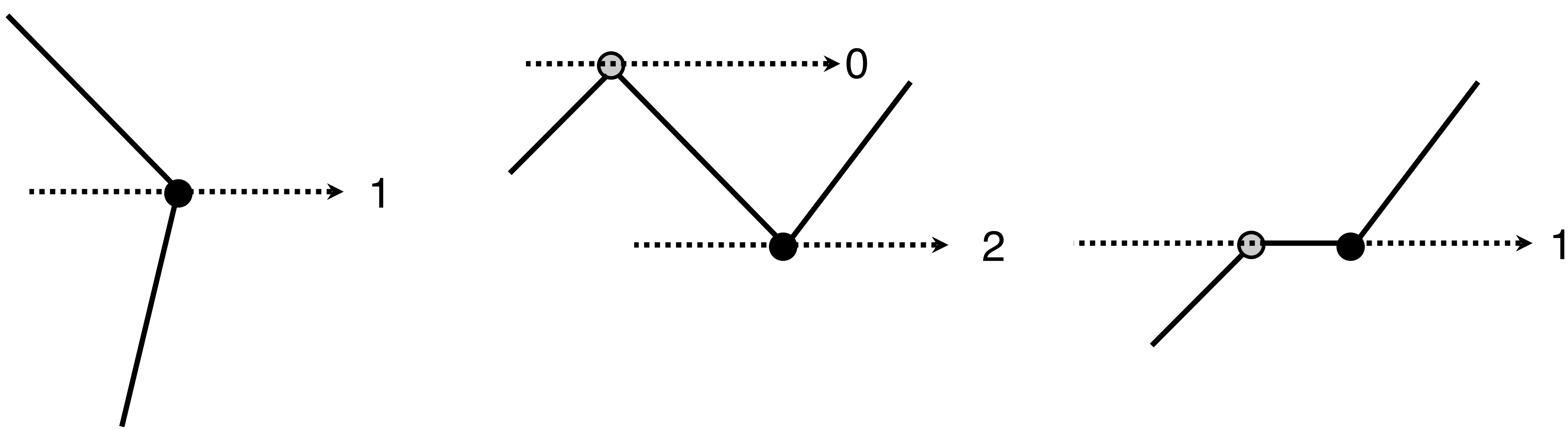
Difficult points



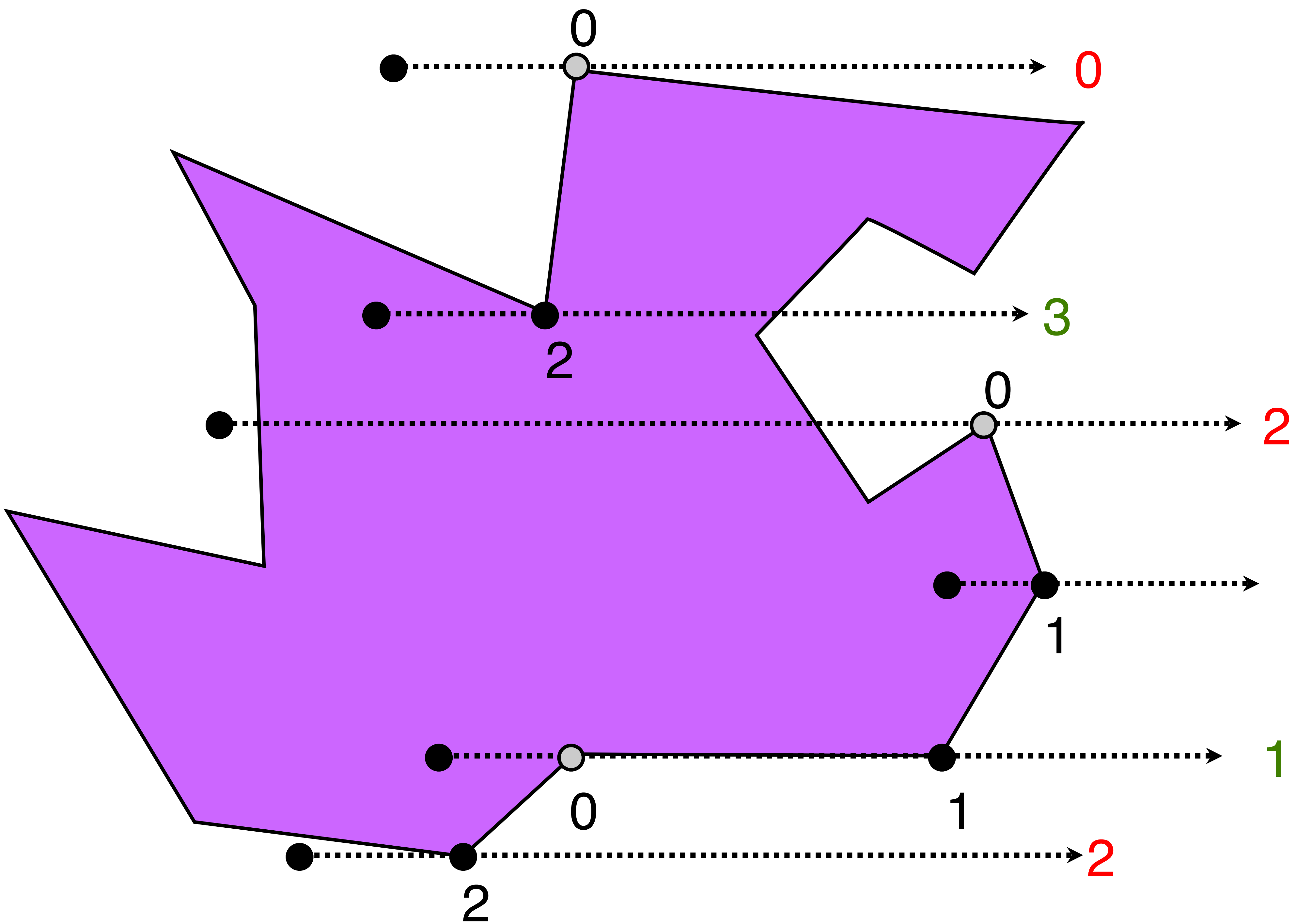
Solution

- Only count crossings at the lower vertex of an edge.





Point in polygon



Computational Geometry

- *Computational Geometry in C*, O'Rourke
- http://cs.smith.edu/~orourke/books/compgeom.html
- CGAL
  Computational Geometry Algorithms Library
- http://cgal.org/

Shaders

- Shaders are programs executed on the GPU for the purpose of rendering graphics.
- They are written in a special language called GLSL (GL Shader Language).

GLSL Syntax

- C like language with
  - No pointers!
  - Basic types: float int bool
  - Standard C/C++ arithmetic and logic operators and overloaded ones to work on vectors and matrices
  - if statements,loops

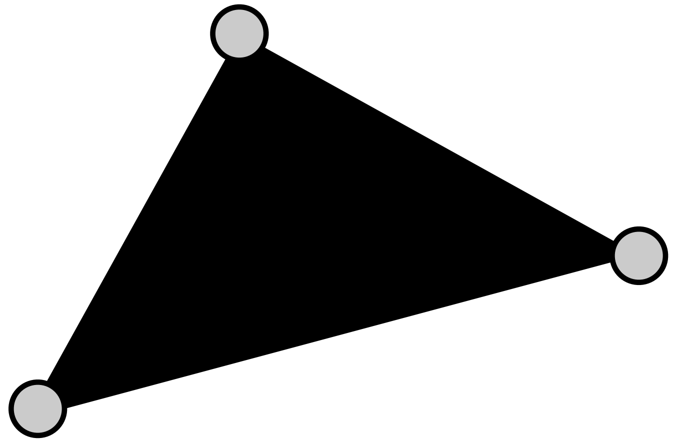
GLSL Syntax

- No characters, strings or printf
  - Hard to debug
- No recursion
- No double (limited support in later versions)

Vertex Shaders

- The GPU will execute the vertex shader for every vertex we supply it.

- e.g. if we're drawing a triangle, the vertex shader will execute three times.



Basic Vertex Shader

- Takes the input position and returns it as is.

```
// Incoming vertex position
in vec2 position;

void main() {
    gl_Position = vec4(position, 0, 1);
}
```

GLSL

- Variables declared:
    - ‘in’ are inputs to the shader and are different for each vertex.
    - ‘uniform’ are inputs to the shader that are the same for every vertex
    - ‘out’ are what the shader outputs
- Variables starting with ‘gl_’ are built-in and have special meaning.

GLSL Syntax

- Has support for 2D, 3D, 4D vectors (array like list like containers) of different types
  - vec2, vec3, vec4 are float vectors
- Operators are overloaded for vector operations.
- Can be constructed in many ways. e.g.

```
vec4 vec4(vec2 xy, float z, float w);
```

GLSL

- gl_Position is a homogenous point in 3D (i.e. a vector of rank 4)

```
// Incoming vertex position
in vec2 position;

void main() {
    gl_Position = vec4(position, 0, 1);
}
```

GLSL

- This shader applies no transformation to the vertex at all (equivalent to UNSWgraph v0.2 and earlier)

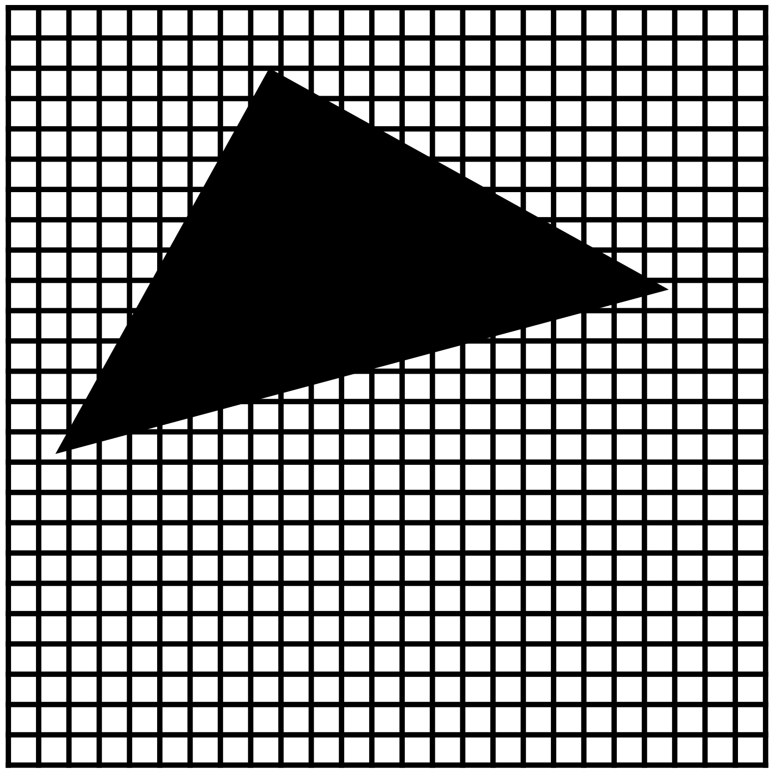
```
// Incoming vertex position
in vec2 position;

void main() {
    gl_Position = vec4(position, 0, 1);
}
```

Fragment Shaders

- The GPU will execute the fragment shader for every pixel it draws into the framebuffer

- e.g. if we're drawing a triangle, the fragment shader will execute for every pixel that gets filled in.



Basic Fragment Shader

- In OpenGL if the fragment shader only has one declared output, then that is what gets written to the framebuffer.

```
out vec4 outputColor;

void main() {
    // Output black
    outputColor = vec4(0,0,0,0);
}
```

Basic Fragment Shader

- For reasons we will cover later, the output is a rank 4 vector. The first 3 components are the RGB values.

```
out vec4 outputColor;

void main() {
    // Output black
    outputColor = vec4(0,0,0,0);
}
```

Setting up Shaders

- OpenGL support a full compiler pipeline for GLSL.
- Shaders can be loaded from text files, compiled, linked and loaded (transferred to the GPU).
- See Shader.java in unsw.graphics

Using Shaders

- After your shaders have been set up you need to tell OpenGL what shaders to use.
- To set the current shader use:

```
gl.glUseProgram(shaderProgramID);
```

- UNSWgraph has Shader.use() that does this

Color Fragment Shader

- This shader allows us to draw in colour

```
out vec4 outputColor;

uniform vec3 input_color;

void main()
{
    // Output whatever was input
    outputColor = vec4(input_color, 0);
}
```

Uniform Variables

- Uniforms are read-only input variables
- Defined in your JOGL program and input into your vertex or fragment shader.
- Can't be changed for a given primitive.

User Defined Uniforms

- To pass in your own uniforms into your shaders from the application program

```
int loc =
gl.glGetUniformLocation(shaderProgram,”myVal”);

gl.glUniform1f(loc,0.5);
```

- Your vertex and/or fragment shader will need a matching declaration like:

```
uniform float myVal;
```

Color Fragment Shader

- See Shader.setPenColor()

```
out vec4 outputColor;

uniform vec3 input_color;

void main()
{
    // Output whatever was input
    outputColor = vec4(input_color, 0);
}
```

Transforming Vertex Shader

```
// Incoming vertex position
in vec2 position;

uniform mat3 model_matrix;

uniform mat3 view_matrix;

void main() {
    // The global position is in homogenous coordinates
    vec3 globalPosition = model_matrix * vec3(position, 1);

    // The position in camera coordinates
    vec3 viewPosition = view_matrix * globalPosition;

    // We must convert from a homogenous coordinate in 2D to a homogenous
    // coordinate in 3D.
    gl_Position = vec4(viewPosition.xy, 0, 1);
}
```

Transforming Vertex Shader

- See shaders/vertex_2d.glsl
- This is how we apply the model and view transforms.
- See Shader.setModelMatrix() and Shader.setViewMatrix()

Matrix Components

- Matrices are in column major order

```
mat4 m = mat4(1.0); //identity matrix
m = mat4(1.0,2.0,3.0,4.0,         //first col
         5.0,6.0,7.0,8.0,         //second col
         9.0,10.0,11.0 12.0,      //third col
         13.0,14.0,15.0,16.0);    //fourth col
float f = m[0][1]; //Would be 2.0
```

GLSL Functions

- Standard Maths functions: sqrt, pow, abs, floor, ceiling, clamp etc
- Trigonometric functions in radians : cos sin tan degrees etc.
- Vector functions: dot, cross, normalize, reflect etc
- Can create user defined functions, syntax similar to C

Exercise

- Write a fragment shader to generate a visualisation of the Mandelbrot set.

- See Mandelbrot.java in unsw.graphics.example.

Revision: Complex numbers

- Have both a real component and an imaginary component.

$$c = a + bi$$

- Where:

$$i = \sqrt{-1}$$

Revision: Complex numbers

- Can be added

$$\begin{aligned} c_1 &= a_1 + b_1 i \\ c_2 &= a_2 + b_2 i \\ c_1 + c_2 &= (a_1 + b_1 i) + (a_2 + b_2 i) \\ &= (a_1 + a_2) + (b_1 + b_2) i \end{aligned}$$

- and multiplied

$$\begin{aligned} c_1 c_2 &= (a_1 + b_1 i)(a_2 + b_2 i) \\ &= a_1 a_2 + a_1 b_2 i + b_1 a_2 i + b_1 b_2 i^2 \\ &= (a_1 a_2 - b_1 b_2) + (a_1 b_2 + b_1 a_2) i \end{aligned}$$

- Note that $i^2 = -1$

Revision: Complex numbers

- The magnitude

$$|a + bi|$$

- is defined as

$$\sqrt{a^2 + b^2}$$

Mandelbrot Set

- Defined as the set of complex numbers, c, for which

$$z_{n+1} = z_n^2 + c$$

remains bounded in its magnitude. i.e if |z| in this loop remains bounded

```
while (true) {
    z = z² + c;
}
```

Mandelbrot Set

- We can't compute it exactly as it requires infinite computation

- We have to approximate it!

- Execute the loop N times, if it doesn't diverge, assume it never will.

- if $|z| > 2$ then it has diverged.