# 3. Geometry

COMP3421 Computer Graphics • KC Notes

## 3.1 Coordinate frames

* 2D frame defined by origin and two axis vectors
* Point can be **described as displacement** from origin
* Point can be **converted to another coordinate frame** using the above equation
* **Affine transformation**: transformation between coordinate frame in matrix form
* **Homogenous coordinates**: extra dimension representing the origin:
  + includes origin (point)
  + does not include origin (vector)
  + Works well with adding a vector and a point becomes a point (1 + 0),   
    and adding two vectors becomes a vector (0 + 0)

## 3.2 Affine transformations

where is a point, a vector

* Matrices in this form (with 0s and 1 at the end of bottom row) are **affine transformations**
  + Can be expressed as combinations of **translation, rotation, scale and shear**
* Properties:
  + Preserves **straight lines and parallel lines**
  + Maintains **relative distance on lines** (e.g. midpoints are still midpoints)
  + Do **not** always **preserve angles or area**
* **2D translation**: translates origin to new point
  + Translating vector has no effect
* **2D rotation**: rotate point about the origin, anticlockwise
* **2D scale**: scale point by factors about the origin
* **Shear**: scale axes non-uniformly, then rotate
  + Does not preserve angles, avoid by scaling uniformly
  + To check for shear, **dot product i and j columns**

Horizontal (bend y towards x axis):

Vertical (bend x towards y axis):

## 3.3 Composing and decomposing transformations

* **Composing transformations**: multiply the matrices, order matters
* **Decomposing transformations**
  + Assuming no shear:

## 3.4 Inverse transformations

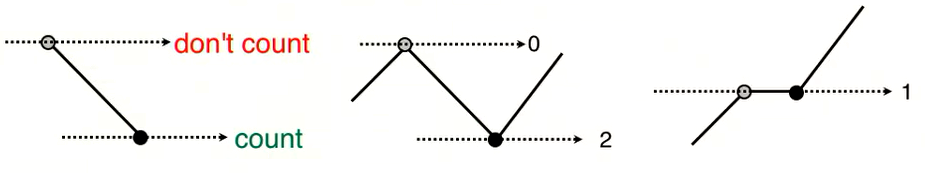
* Translation:
* Rotation:
* Scale:
* Shear:

## 3.5 Lines

* **Linear interpolation** between points – (useful for obtaining an in-between point)
* For example, midpoint:
* **Point-normal form** (useful for checking if point is on line):

## 3.6 Point in polygon

* **Intersection**: solve simultaneous equations, split into top and bottom, solve, then sub in t.
* To check if **point is within polygon**, draw a line to the right across the polygon
  + If passes **even number**, outside polygon
  + If passes **odd number**, inside polygon
  + Only count crossing at the lower vertex of an edge



## 3.7 Vertex shaders

* **Shaders**: programs executed on GPU to generate graphics, in GLSL (GL Shader Language)
* OpenGL is secretly a compiler – supports a full compiler pipeline for GLSL
* Shaders loaded from text files, compiled, linked and loaded (transferred to GPU)
  + Use via gl.glUseProgram(id); - for multiple shaders, change the ID
* Errors not picked up till runtime
* **Vertex shader**: execute **for every vertex** we supply it
  + If drawing a triangle, vertex shader executes three times

|  |  |
| --- | --- |
| // Incoming vertex position  in vec2 position;  uniform mat3 model\_matrix;  uniform mat3 view\_matrix;  void main() {  // Global position in homogenous coordinates  vec3 globalPosition = model\_matrix \*   vec3(position, 1);  // Position in camera coordinates  vec3 viewPosition = view\_matrix \* globalPosition;  // Convert from a homogenous coordinate in 2D to 3D  gl\_Position = vec4(viewPosition.xy, 0, 1);  } | Convert from local to global coordinates  Convert from global to camera coordinates |
|  |  |

* Basic GLSL:
  + float, int, bool
  + if statements and loops, no recursion
  + vec2, vec3, vec4 are float vectors, can be constructed
  + sqrt, pow, abs, floor, ceiling, clamp
  + cos, sin, tan, degrees
  + dot, cross, normalize, reflect
* Variables:
  + in: inputs to the shader, different for each vertex
  + uniform: inputs that are the same for each vertex – read-only
  + out: shader outputs
  + gl\_: built-in variables with special meaning
* **Setting model\_matrix, view\_matrix**:

|  |  |
| --- | --- |
| public static void setModelMatrix(GL3 gl, Matrix3 mat) {  int ids[] = new int[1];  gl.glGetIntegerv(GL3.GL\_CURRENT\_PROGRAM, ids, 0);  int modelLoc = gl.glGetUniformLocation(ids[0], "model\_matrix");  gl.glUniformMatrix3fv(modelLoc, 1, false, mat.getValues(), 0);  } | |
| public static void setFloat(GL3 gl, String var, float f) {  int ids[] = new int[1];  gl.glGetIntegerv(GL3.GL\_CURRENT\_PROGRAM, ids, 0);  int loc = gl.glGetUniformLocation(ids[0], var);  gl.glUniform1f(loc, f);  } | |
|  |  |

## 3.8 Fragment shaders

* GPU executes fragment shader **for every pixel it draws** in the frame buffer
  + Fragment shader executes for every pixel that of a triangle that gets filled in
  + Determines what colour it should be

|  |  |
| --- | --- |
| out vec4 outputColor;  uniform vec3 input\_color;  void main() {  // Output whatever was input  outputColor = vec4(input\_color, 0);  } | |
|  |  |

## 3.9 Mandelbrot shader

* Outputs of vertex shader becomes input of fragment shader
  + Inputs are interpolated between the vertices
    - outputColor = vec4(globalPosition.xy, 0, 1);

|  |
| --- |
| out vec4 outputColor;  uniform vec4 input\_color;  in vec3 globalPosition;  void main() {  // Output whatever was input  vec4 color = input\_color;  float R = globalPosition.x;  float Im = globalPosition.y;  int i = 0;  for (i = 0; i < 128; i++) {  float R2 = R\*R - Im\*Im + globalPosition.x;  float Im2 = 2\*R\*Im + globalPosition.y;  if (R2\*R2 + Im2\*Im2 > 4) break;  R = R2;  Im = Im2;  }  // fract gets fractional component of the number  float i2 = (128-i)/127.0;  outputColor = vec4(i2, fract(i2\*2), fract(i2\*3), 1);  } |