# 2. Transforms

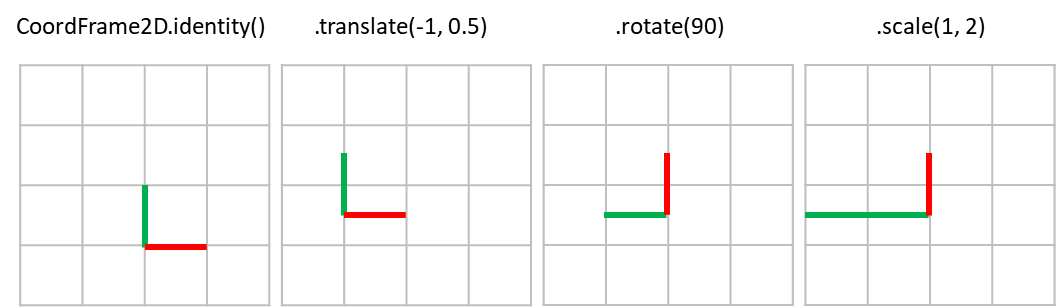
COMP3421 Computer Graphics • KC Notes

## 2.1 Transformations

* Types of transformations
  + Translate – moves origin of the coordinate frame
  + Rotate – rotates around the origin, x to y/anticlockwise
  + Scale across different axes – scales coordinate space in x or y, allows reflections
  + If object is away from the origin, there may be some undesired effects
    - Rotation is off by a distance, scaling will increase distance from origin
* Transformations done in two ways: **extrinsic and intrinsic**
  + **Extrinsic**: object being transformed or altered within a **fixed coordinate system**
  + **Intrinsic**: coordinate system of the object is transformed
    - Do not want to rewrite buffer every time, so best change the coordinate frame and then draw the object into the frame without changing buffer
* **Model transformation**: how a **local** coordinate system maps to the **world** coordinate system
  + Each object in a scene has its own local coordinate system

## 2.2 Coordinate frames

* Coordinate system is defined by a **coordinate frame**
* Defined by **origin, direction and scale of x and y axes**
* **Identity frame**: a frame with origin at (0, 0), x-axis horizontal, y-axis vertical, both of length 1

****

* **Order of transformations** matters
  + Translate then rotate != rotate then translate
  + Translate then scale != scale then translate (scaled frame will move further)
  + Rotate then scale != scale then rotate (angles not preserved when x and y are scaled to different amounts – shear transform)

## 2.3 Other transformations

* Rotate around arbitrary point
  + Translate to point, rotate, then translate back
    - Translate moves the origin
* Storing frame history
  + CoordFrame2D is immutable – can store intermediate frames
  + CoordFrame2D f1 = CoordFrame2D.identity().scale(0.5f, 0.5f);  
    CoordFrame2D f2 = f1.translate(-1, 1);

## 2.4 Scene tree

* **Instance Transformation**: every object has translation, rotation and scale,
* Use successive transformations for each object’s frame to position an object
  + Objects do not know about parent’s position or transformation, only its frame
* Be aware of **reparenting** – need to convert back to global then back to new local coords

|  |  |
| --- | --- |
| drawTree(frame) {  compute new\_frame by transforming frame:  translation  rotation  scale  draw the object  for all children:  child.drawTree(new\_frame) | |
| public void draw(GL3 gl, CoordFrame2D frame) {  CoordFrame2D objectFrame = frame  .translate(myTranslation)  .rotate(myRotation)  .scale(myScale, myScale);  drawSelf(gl, objectFrame);  for (SceneObject child : myChildren) {  child.draw(gl, objectFrame);  }  } | |
|  |  |

* If person’s frame is scaled, it will look squished, not shorter
* If we want to shrink the torso without squishing the head:
  + Have a **pivot element at root** instead of the torso
  + Torso and head are separately under the pivot element, scale torso

## 2.5 Circles

|  |  |
| --- | --- |
| ArrayList<Point2D> points = new ArrayList<Point2D>();  for (int i = 0; i < CIRCLE\_SEGMENTS; i++) {  float x = radius \* (float)Math.cos(Math.toRadians(i / SEGMENTS \* 360));  float y = radius \* (float)Math.sin(Math.toRadians(i / SEGMENTS \* 360));  points.add(new Point2D(x, y));  }  return points; | |
|  |  |

## 2.6 Camera

* When camera is moved, imagine the world is moved instead
* Camera can also be an object in the scene tree
  + World must be **rendered as it appears in the camera’s local coordinate frame**



* **Model transform**: convert object points in **local to world** coordinates
* **View transform**: convert **world** frame to **camera’s** **local** coordinate frame
  + Transforms points in the world to points in camera

|  |  |
| --- | --- |
| // Sets the view transform (world to camera coordinates)  // Allows camera to be moved around the scene without affecting camera  Matrix3 viewTransform = Matrix3.identity()  .multiply(Matrix3.scale(1/aspectRatio, 1))  .multiply(Matrix3.scale(1/getGlobalScale(), 1/getGlobalScale()))  .multiply(Matrix3.rotation(-getGlobalRotation()))  .multiply(Matrix3.translation(-getGlobalPosition().getX(),  getGlobalPosition().getY()));  Shader.setViewMatrix(gl, viewTransform); | |
| // Transforms a point from camera to world coordinates  Matrix3 mat = Matrix3.translation(getGlobalPosition())  .multiply(Matrix3.rotation(getGlobalRotation()))  .multiply(Matrix3.scale(getGlobalScale(), getGlobalScale()))  .multiply(Matrix3.scale(myAspectRatio, 1));  return mat.multiply(new Vector3(x,y,1)).asPoint2D(); | |
|  |  |

## 2.7 Vectors and Matrices

* **Vector**: displacement, tuple of values, **length and direction**, no position
* **Point**: **position**, no length or direction
* Magnitude:
* Normalisation:
* Dot product:
* Properties:
  + **Angle between vectors**:
    - For two perpendicular vectors, , then
* Cross product: (wrap a to b, thumb points up at cross product)
* Properties:
  + **Magnitude of cross product is area of parallelogram**:
* Matrices: multiplication

## 2.8 Color buffer

* Holds colour information about the pixels
* Needs to be cleared on every frame – Application2D.display() does this

|  |  |
| --- | --- |
| // Set the clear color  gl.glClearColor(getBackground().getRed(), getBackground.getGreen(),  getBackground.getBlue(), getBackground.getAlpha());  // Clear screen with defined clear color  gl.glClear(GL.GL\_COLOR\_BUFFER\_BIT); | |
|  |  |

* **Single buffering**: one buffer being both drawn to and sent to the monitor. Flickering.
* **Double buffering**: uses two buffers, draw into back buffer while the front buffer is displayed. Swap buffers after updating is finished (default in OpenGL)

## 2.9 Animation

* Keep track of displacement of fish
* Keep track of time since last draw