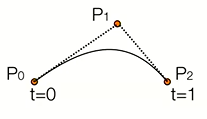
# 7. Modelling and Curves

COMP3421 Computer Graphics • KC Notes

## 7.1 Curves

* Curves expressed parametrically, e.g.
* Curves can be expensive – e.g. trigonometric operations, reduce floating point operations, but allows intuitive curve design – **interpolating control points** is a good solution

## 7.2 Interpolation

* Linear interpolation: linear function of **degree 1**, with 2 control points (**order 2**)
* Quadratic interpolation:
  + **Interpolates** passes through and
  + **Approximates** passes near
  + **Tangents** at and point to
* Cubic interpolation: similar

## 7.3 de Casteljau algorithm

* Possible to do **quadratic interpolation in 3 linear steps**
* Possible to do **cubic interpolation in 4 linear steps + 3 steps**
* Family of curves are known as **Bezier curves** with general form

where is the **degree** of the curve, are the **control points**,

* Bezier curves interpolate endpoints, approximate all intermediate points
  + Are **convex combinations** of points
  + Invariant under transformation –**curve is based on transformed control points**
    - Transforming this control points = transforming entire curve itself
  + Curve lies within **convex hull** of control points, e.g. it will fit in a box

## 7.4 Tangent to the curve

* **Tangent vector** to curve – it is a curve of degree
  + i.e. Take the of points, linearly interpolate between the points and multiply by m

## 7.5 Problems

* Degree of Bernstein polynomials is coupled to number of control points – L+1 control points is a combination of L-degree polynomials
  + High degree is more expensive, vulnerable to rounding errors
* **Non-local control** – moving one point affects the whole curve – **splines**
* Spline: curves joined together, smooth piecewise-polynomial function that **allows local control**
  + Places where they join are **knots**
  + Joining them together it may not be smooth

## 7.7 3D Modelling

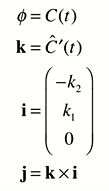
* Simple Revolution
  + Compute a triangle mesh and rotate around by a number of slices of the circle
    - Indices: add i’th, i+1th and centre (0)
    - Normals: (x, y, 0) – same as the point on the circle
    - Shared vertex is strange, so duplicate the shared vertices

|  |  |
| --- | --- |
| private TriangleMesh makeCone(GL3 gl) {  // Make the approximating triangular mesh.  List<Point3D> vertices = new ArrayList<Point3D>();  List<Vector3> normals = new ArrayList<Vector3>();  List<Integer> indices = new ArrayList<Integer>();    float tIncrement = 1f/NUM\_SLICES;  for (int i = 0; i < NUM\_SLICES; i++) {  float t = i\*tIncrement;  vertices.add(new Point3D(0, 0, height));    float x = getX(t);  float y = getY(t);    normals.add(new Vector3(x, y, radius/height));  }    for(int i = 0; i < NUM\_SLICES; i++) {  float t = i\*tIncrement;  float x = getX(t);  float y = getY(t);  vertices.add(new Point3D(getX(t), getY(t), 0));  normals.add(new Vector3(x, y, radius/height));  indices.add(i+NUM\_SLICES);  indices.add((i+1) % NUM\_SLICES + NUM\_SLICES);  indices.add(i);  }  TriangleMesh cone = new TriangleMesh(vertices, normals, indices);  cone.init(gl);  return cone;  } | |
|  |  |

* Extrusion: make a copy and create rectangles along the extruded area
  + Make a prism – translate to another location and **draw triangles in reverse order**
  + Collections.reverse(shapeExtIndices)
  + Sides share vertices, but should not share normals

|  |
| --- |
| private void makeExtrusion(GL3 gl) {  //The **initial shape we're extruding**  List<Point3D> shape = Arrays.asList(new Point3D(0,0,0),...  );  // **Indices** to draw the shape  List<Integer> shapeIndices = Arrays.asList(0,1,6, //indices of the points for the triangle part  2,3,4,4,5,2); //indices for the quad part  // The **initial shape as its own mesh**  TriangleMesh front = new TriangleMesh(shape, shapeIndices, true);    Matrix4 m = Matrix4.translation(0, 0, -1).multiply(Matrix4.scale(2, 2, 1)  .multiply(Matrix4.rotationZ(90)));  // The **extruded shape**  List<Point3D> shapeExt= new ArrayList<>();  for (Point3D p : shape)  shapeExt.add(m.multiply(p.asHomogenous()).asPoint3D());    **// Indices for the extruded shape**  List<Integer> shapeExtIndices = new ArrayList<>(shapeIndices);  Collections.reverse(shapeExtIndices);    **// The extruded shape as its own mesh**  TriangleMesh back = new TriangleMesh(shapeExt, shapeExtIndices, true);    // We want the sides to have their own normals, so we copy the front and  // back faces  List<Point3D> sides = new ArrayList<>();  List<Integer> sideIndices = new ArrayList<>();  for (int i = 0; i < 7; i++) {  //The corners of the quad we will draw as triangles  Point3D bl = shape.get(i);  Point3D br = shapeExt.get(i);  Point3D tl = shape.get((i+1) % 7);  Point3D tr = shapeExt.get((i+1) % 7);    //First triangle  sides.add(bl); ...  // Second triangle ...  //Indices  }    TriangleMesh sidesMesh = new TriangleMesh(sides, true);    front.init(gl);  back.init(gl);  sidesMesh.init(gl);  meshes.add(front); meshes.add(back); meshes.add(sidesMesh);  } |

## 7.8 Segmented Extrusion

* Can continue extruding – **segmented extrusions, transform segments in different matrices**
  + Figure out the spine/curve, compute values (sample ts) along the curve
  + Generate transformation matrix for that section
  + Multiply each point on cross section by the matrix
  + Join these points to the next set of points using quads or triangles
* **Frenet Frame**
  + Define coordinate frame along the curve such that the z axis is a **tangent to the curve**
  + Take tangent and build frame around them
  + Equations on the left define the frame – k is made to be the tangent to the curve , choose I and j to be perpendicular
    - i takes k axis and rotates 90 degrees
* To find the **tangent**
  + Use maths, e.g.
  + Approximate the tangent

## 7.9 Revolution

* Martini glass: for each slice do this:

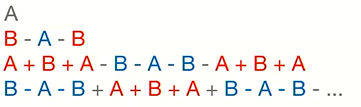
|  |
| --- |
| float theta = 360\*j/(float) NUM\_SLICES;  float x1=(float) (x[i]\*Math.cos(theta\*2\*Math.PI/360f));  float x2=(float) (x[i+1]\*Math.cos(theta\*2\*Math.PI/360f));  float z1=(float) (x[i]\*Math.sin(theta\*2\*Math.PI/360f));  float z2=(float) (x[i+1]\*Math.sin(theta\*2\*Math.PI/360f));  points.add(new Point3D(x1, y[i], z1));  points.add(new Point3D(x2, y[i+1], z2));  // The index of the next slice  int k = (j + 1) % NUM\_SLICES;  indices.add(2\*j);  indices.add(2\*j + 1);  indices.add(2\*k + 1);  indices.add(2\*j);  indices.add(2\*k + 1);  indices.add(2\*k); |

## 7.10 L-Systems

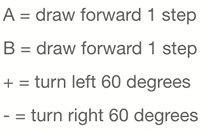
* **Lidenmayer System**: method for producing fractal structures – initially for modelling plant growth with a certain depth
* **A rewrite system** where you keep replacing things

Symbols: A, B, +, -

Rules:



Draw by applying turtle graphics commands:



* Can apply parameters, e.g. , :draw forward s units, or
* Push and pop where [ is push, ] is pop
* Stochastic production systems: random rules, e.g. 50% chance to change to one thing