COMP3421

Modeling, Bezier Curves, L-Systems, VBOs

Curves

We want a general purpose solution for drawing curved lines and surfaces. It should:

- Be easy and intuitive to draw curves
- Support a wide variety of shapes, including both standard circles, ellipses, etc and "freehand" curves.
- Be computationally cheap.

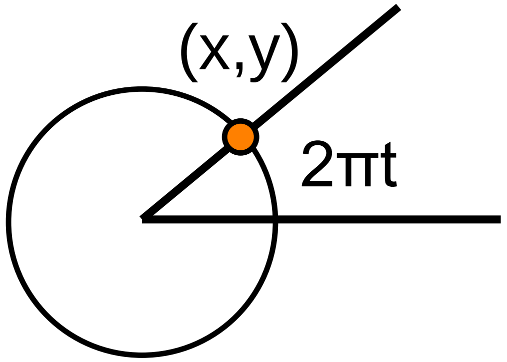
Parametric curves

It is generally useful to express curves in parametric form:

$$\begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} = P(t), \text{ for } t \in [0,1]$$

Eg:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos 2\pi t \\ \sin 2\pi t \end{pmatrix}$$



Interpolation

Trigonometric operations like sin() and cos() are expensive to calculate.

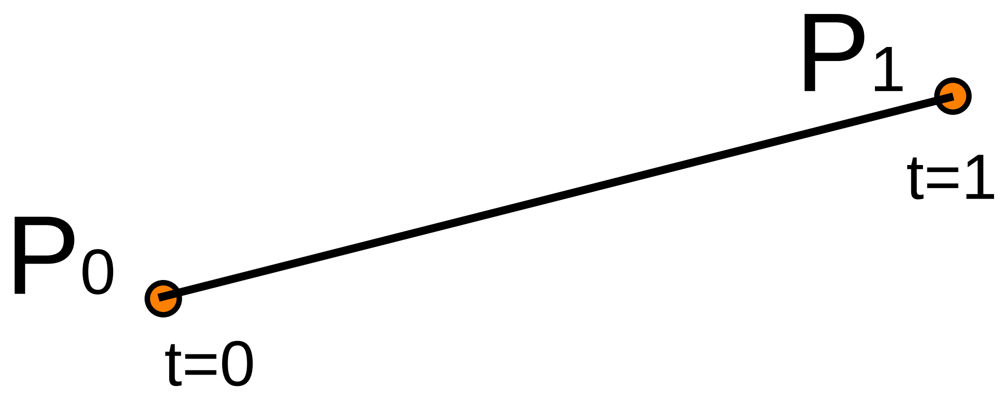
We would like a solution that involves fewer floating point operations.

We also want a solution which allows for intuitive curve design.

Interpolating control points is a good solution to both these problems.

Linear interpolation

$$P(t) = (1-t)P_0 + tP_1$$



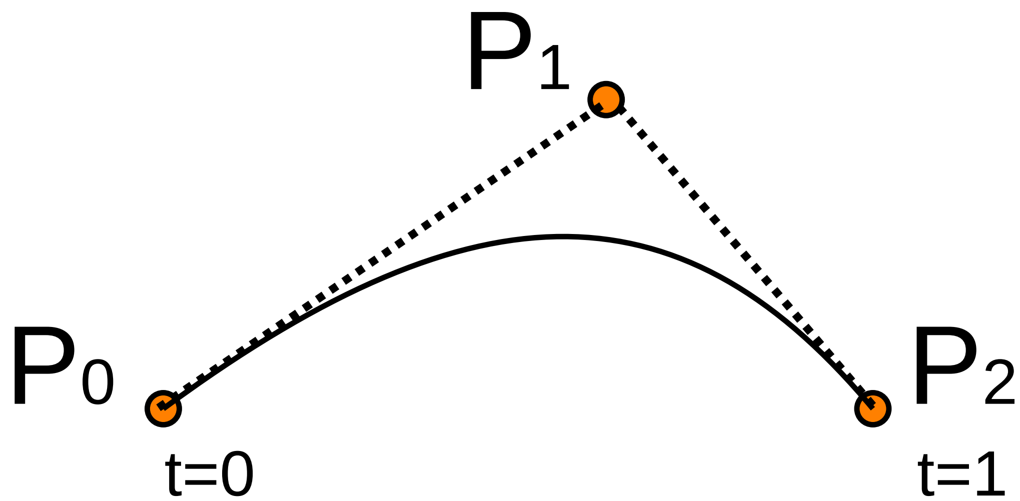
Good for straight lines.

Linear function: Degree 1

2 control points: Order 2

Quadratic interpolation

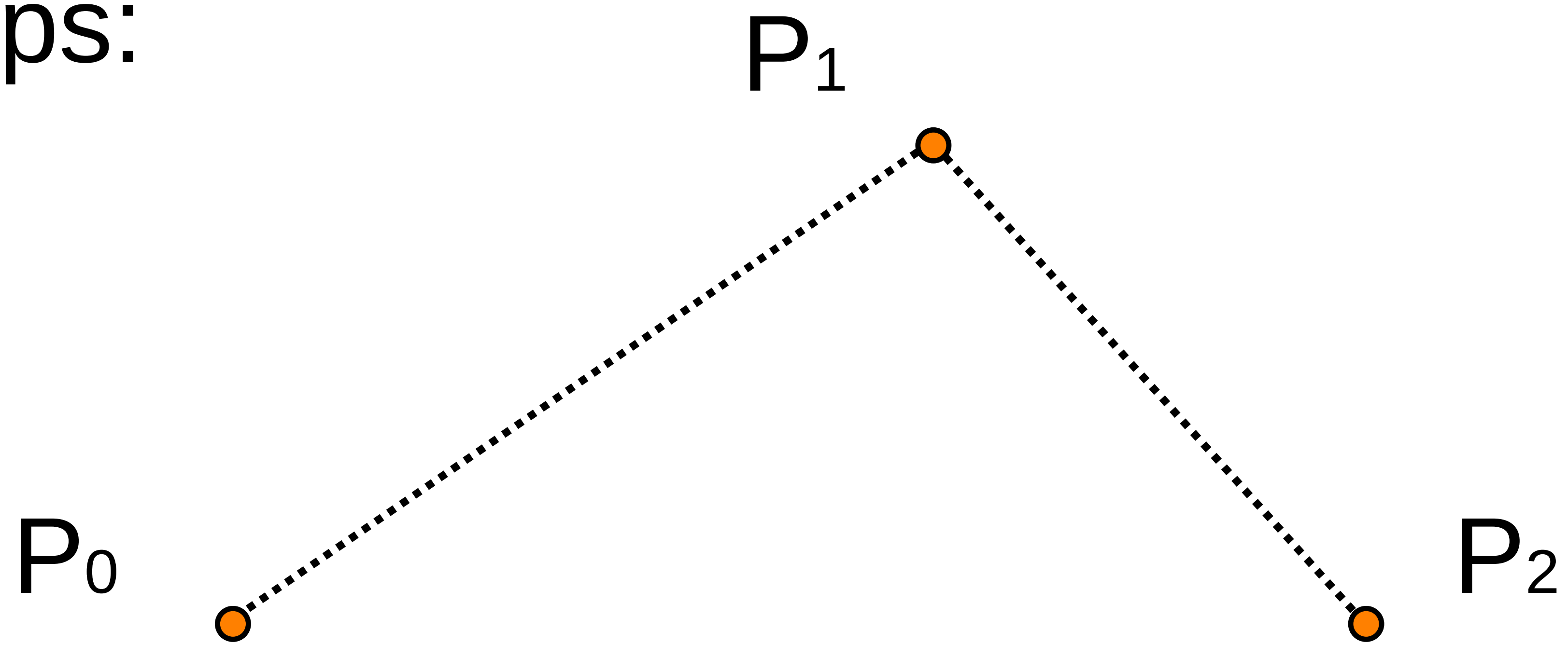
$$P(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2$$



Interpolates (passes through) P0 and P2.
Approximates (passes near) P1.
Tangents at P0 and P2 point to P1.
Curves are all parabolas.

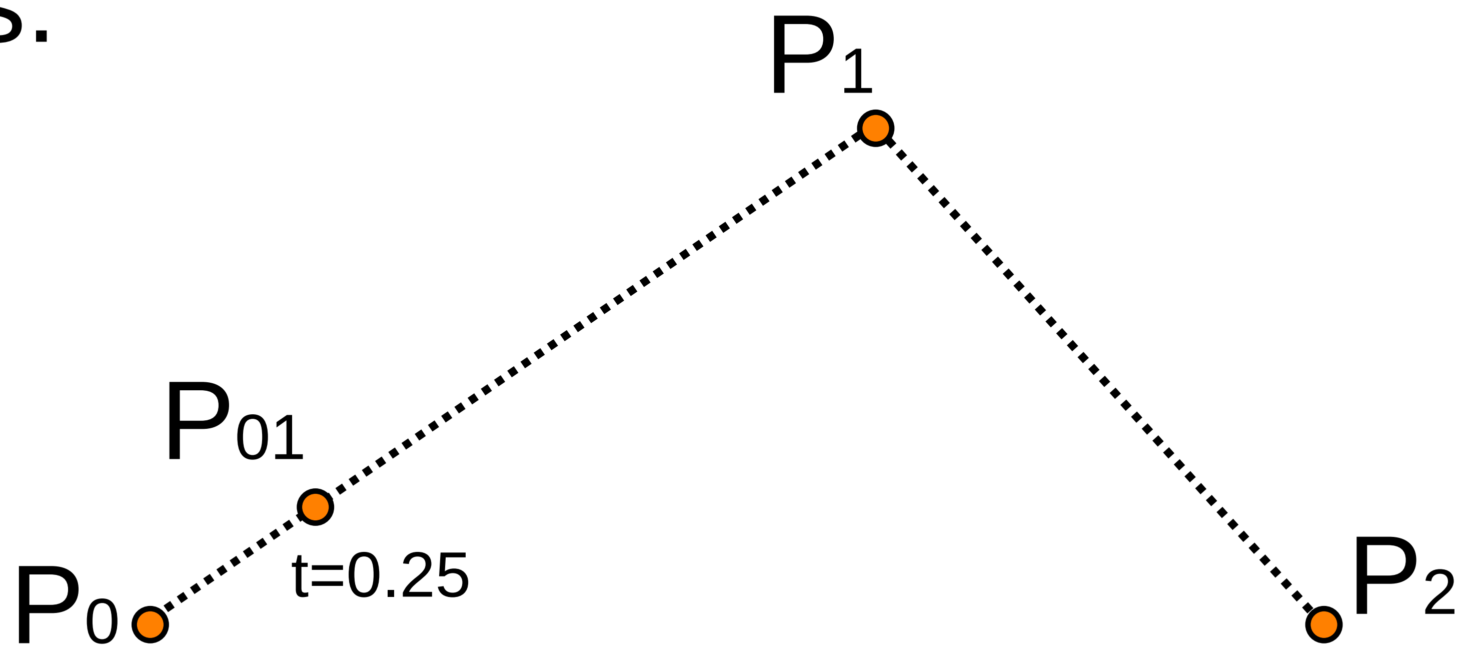
de Casteljau Algorithm

The quadratic interpolation above can be computed as three linear interpolation steps:



de Casteljau Algorithm

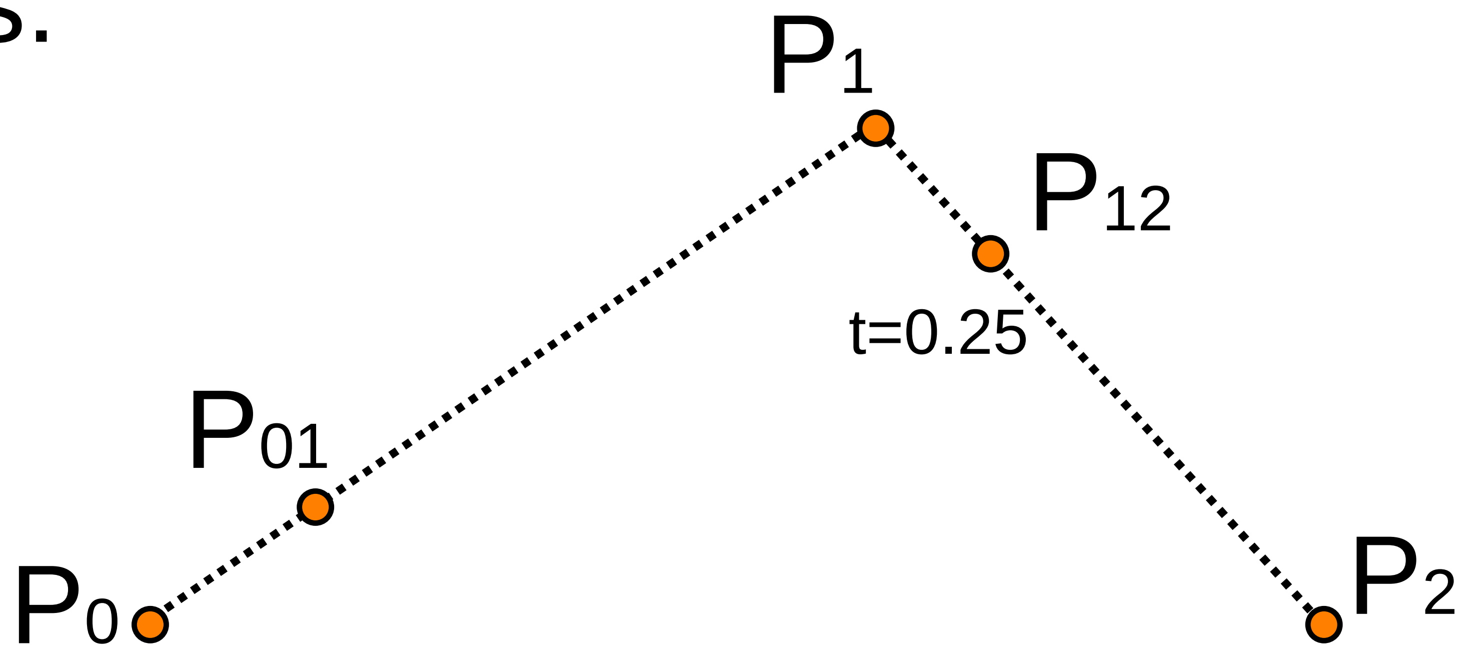
The quadratic interpolation above can be computed as three linear interpolation steps:



$$P_{01}(t) = (1-t)P_0 + tP_1$$

de Casteljau Algorithm

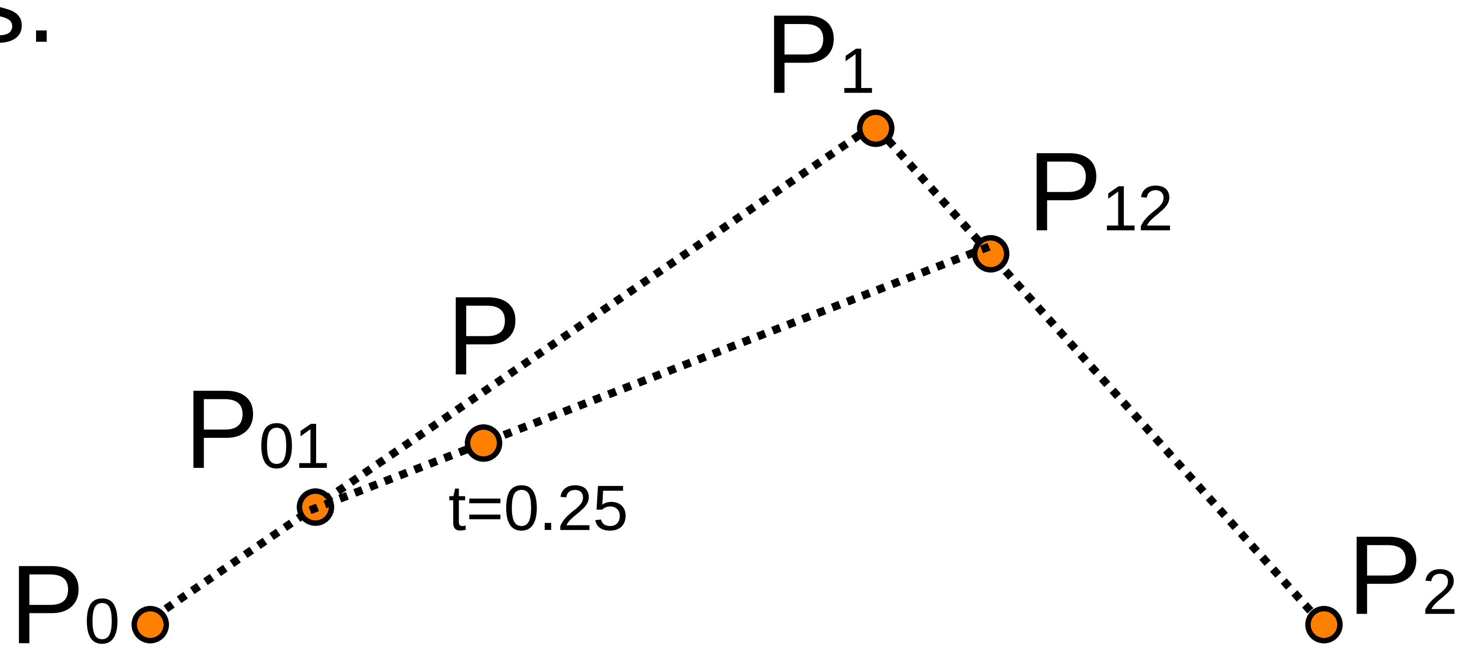
The quadratic interpolation above can be computed as three linear interpolation steps:



$$P_{12}(t) = (1-t)P_1 + tP_2$$

de Casteljau Algorithm

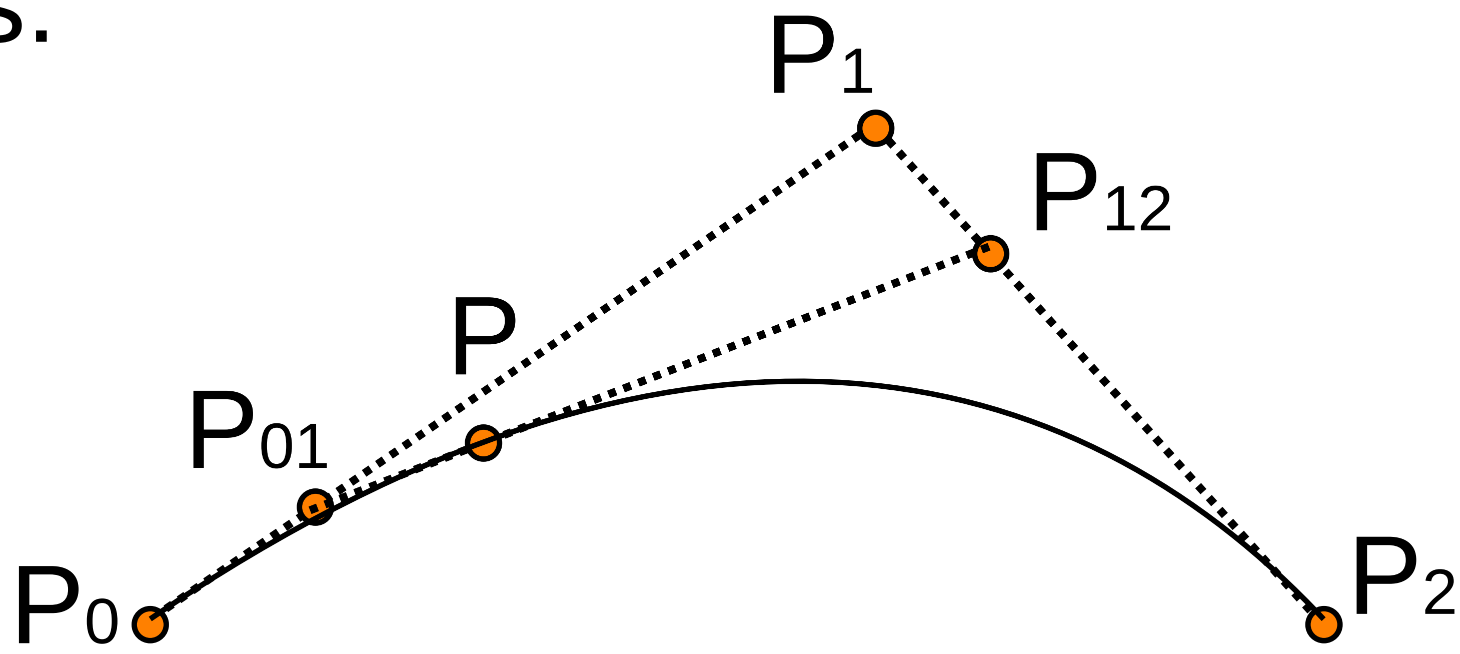
The quadratic interpolation above can be computed as three linear interpolation steps:



$$P(t) = (1-t)P_{01} + tP_{12}$$

de Casteljau Algorithm

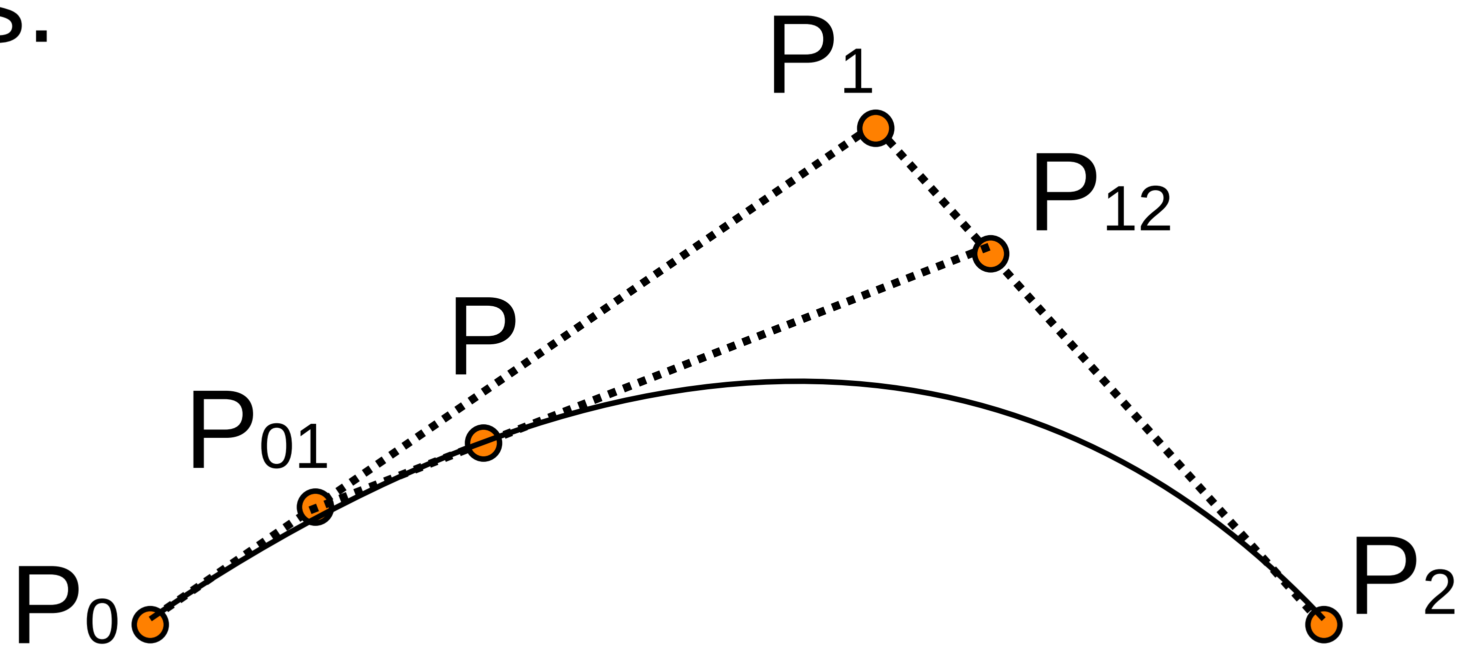
The quadratic interpolation above can be computed as three linear interpolation steps:



$$P(t) = (1 - t)P_{01} + tP_{12}$$

de Casteljau Algorithm

The quadratic interpolation above can be computed as three linear interpolation steps:



$$P(t) = (1-t)P_{01} + tP_{12}$$

$$P(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2$$

de Casteljau Algorithm

$P_{01}(t) = (1\text{-}t)P_0 + tP_1$

$P_{12}(t) = (1\text{-}t)P_1 + tP_2$

$P(t) = (1\text{-}t)P_{01} + tP_{12}$

$= (1\text{-}t)\ ((1\text{-}t)P_0 + tP_1) + t((1\text{-}t)P_1 + tP_2))$

$= (1\text{-}t)\text{^}2P_0 + 2t(1\text{-}t)P_1 + t\text{^}2P_2$

Exercise

Using de Casteljau's algorithm calculate the point at t = 0.75 for the quadratic Bezier with the following control points.

(0,0) (4,8) (12,4)

Confirm your answer using the equation

$$P(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2$$

Exercise Solution

$P_{01}(0.75) = (0.25)(0,0) + 0.75(4,8) = (3,6)$

$P_{12}(0.75) = (0.25)(4,8) + 0.75(12,4)$

$= (1,2) + (9,3) = (10,5)$

$P_{012}(0.75) = (0.25)P_{01} + 0.75P_{12}$

$= (0.25)(3,6) + 0.75(10,5)$

$= (0.75, 1.25) + (7.5, 3.75)$

$= (8.25, 5.25)$

Exercise Solution

Or by using the final formula instead:

$P(0.75) = (1\text{-}t)\text{^}2P_0 + 2t(1\text{-}t)P_1 + t\text{^}2P_2$

$= 0.25\text{^}2(0,0) +$

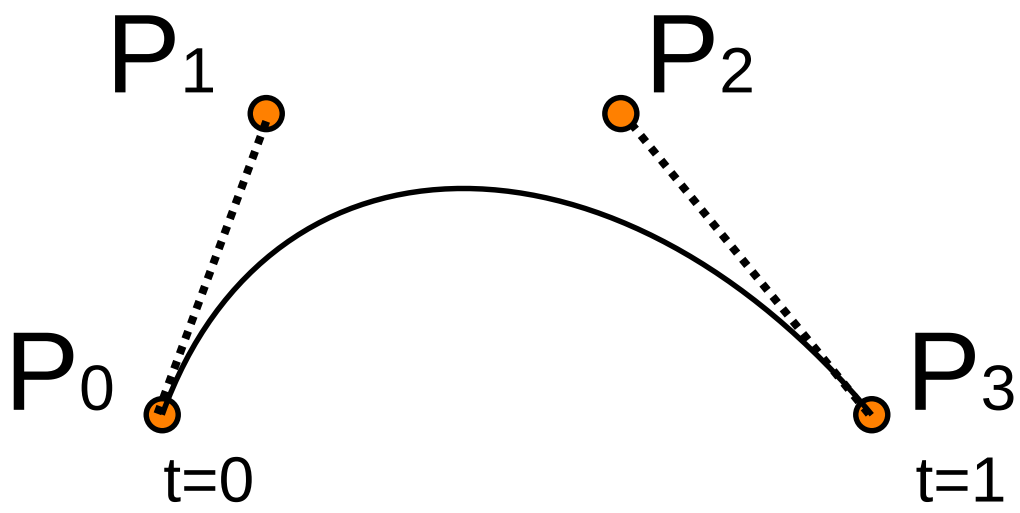
$2 * 0.75 * 0.25\ (4,8) +$

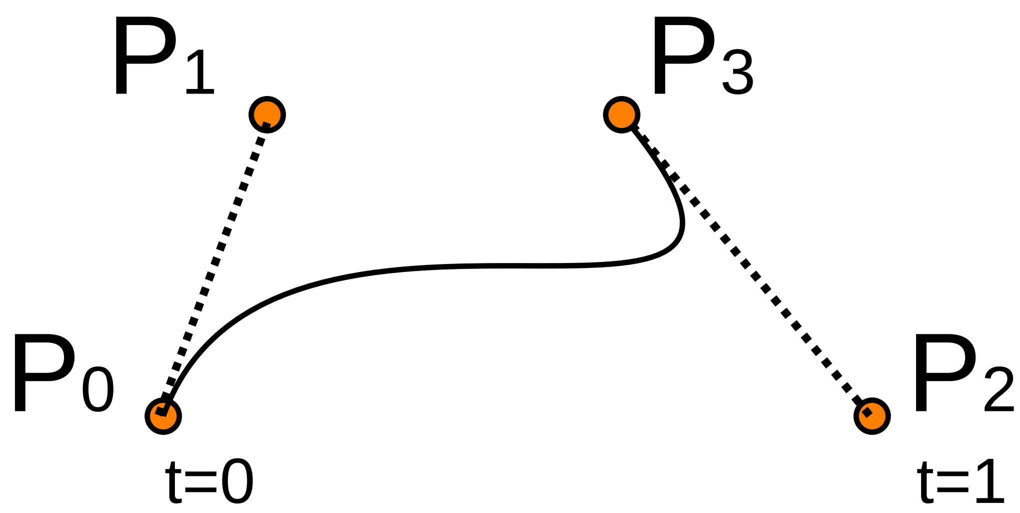
$0.75\text{^}2\ (12,4)$

$= (8.25, 5.25)$

Cubic interpolation

$$P(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t) P_2 + t^3 P_3$$





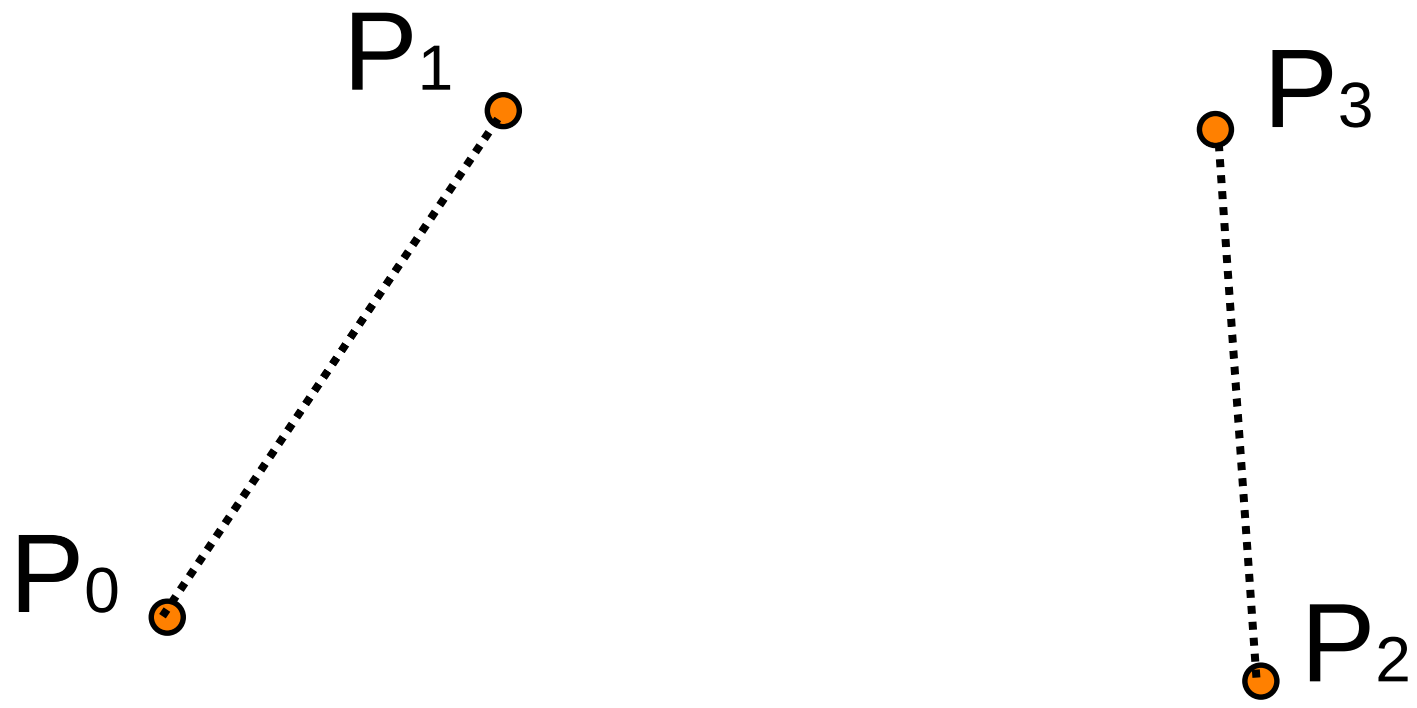
Interpolates (passes through) P0 and P3.

Approximates (passes near) P1 and P2.

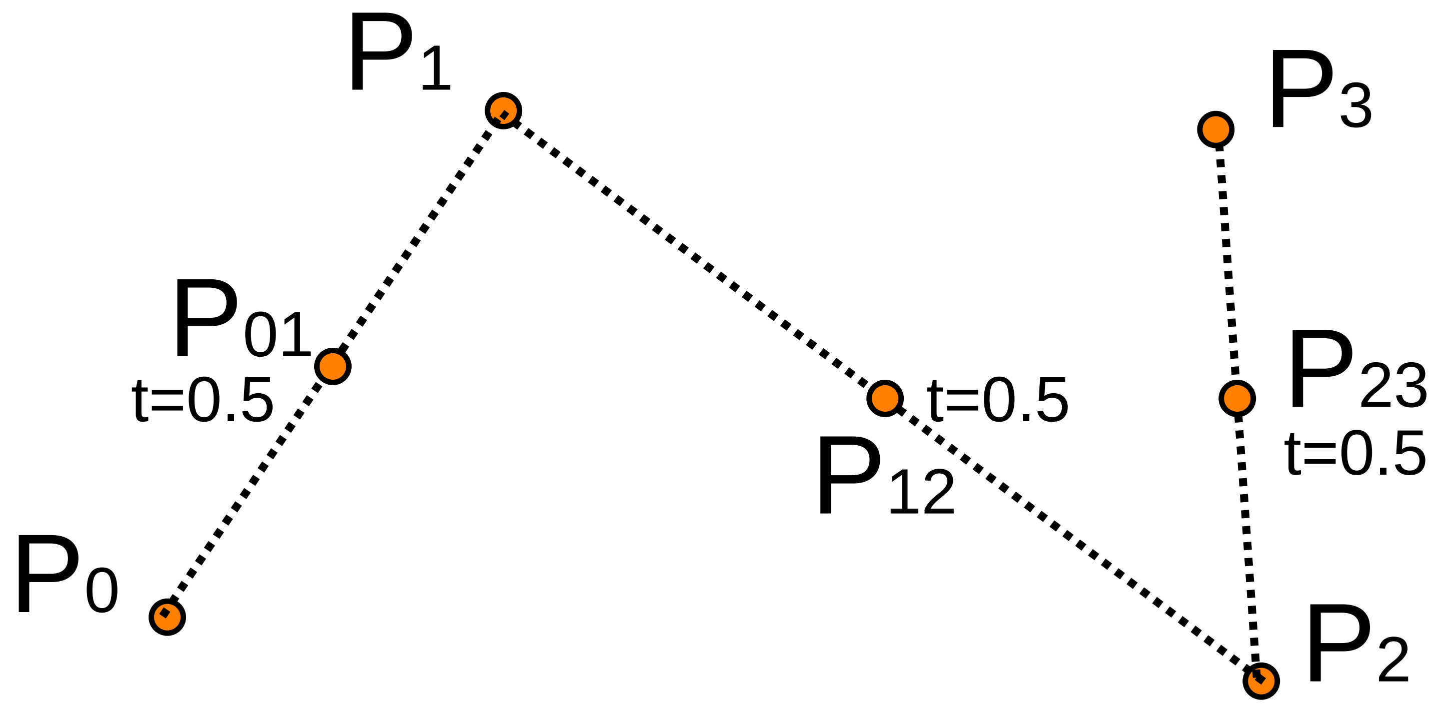
Tangents at P0 to P1 and P3 to P2.

A variety of curves.

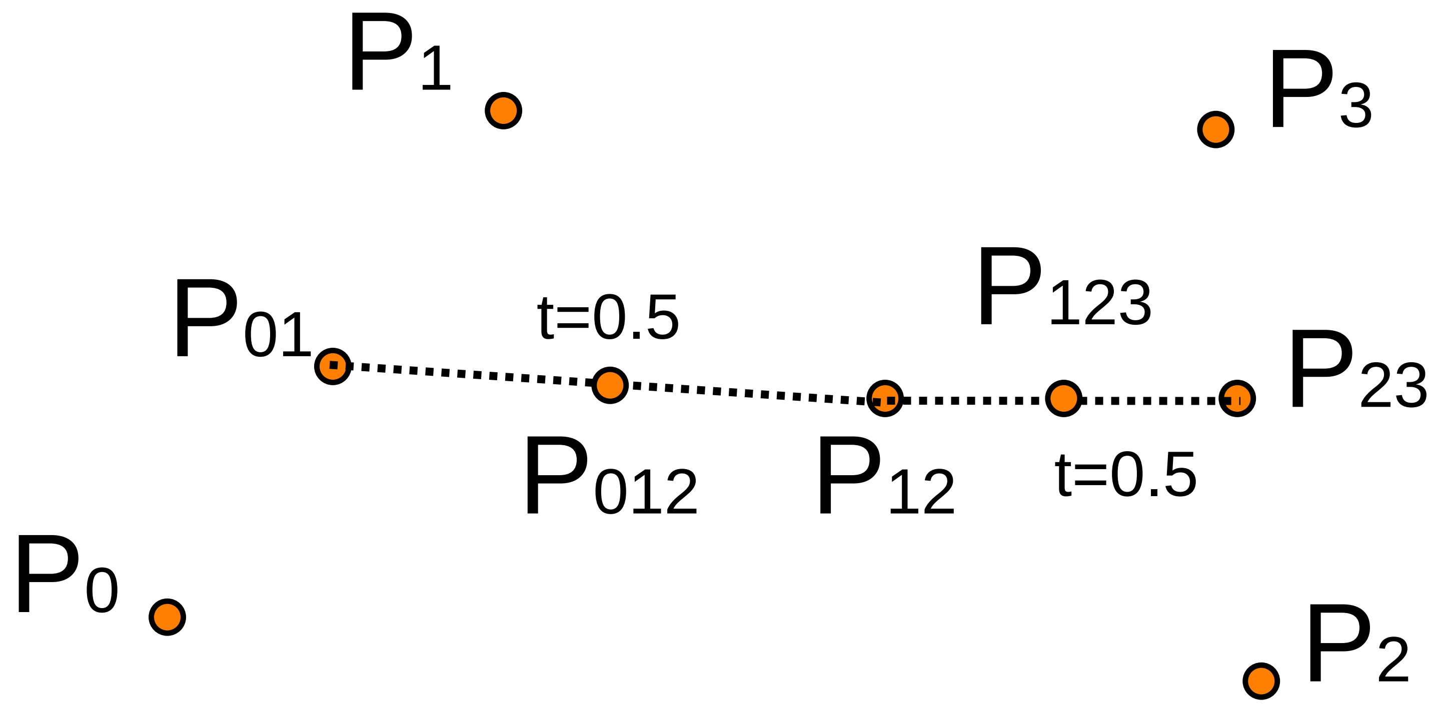
de Casteljau



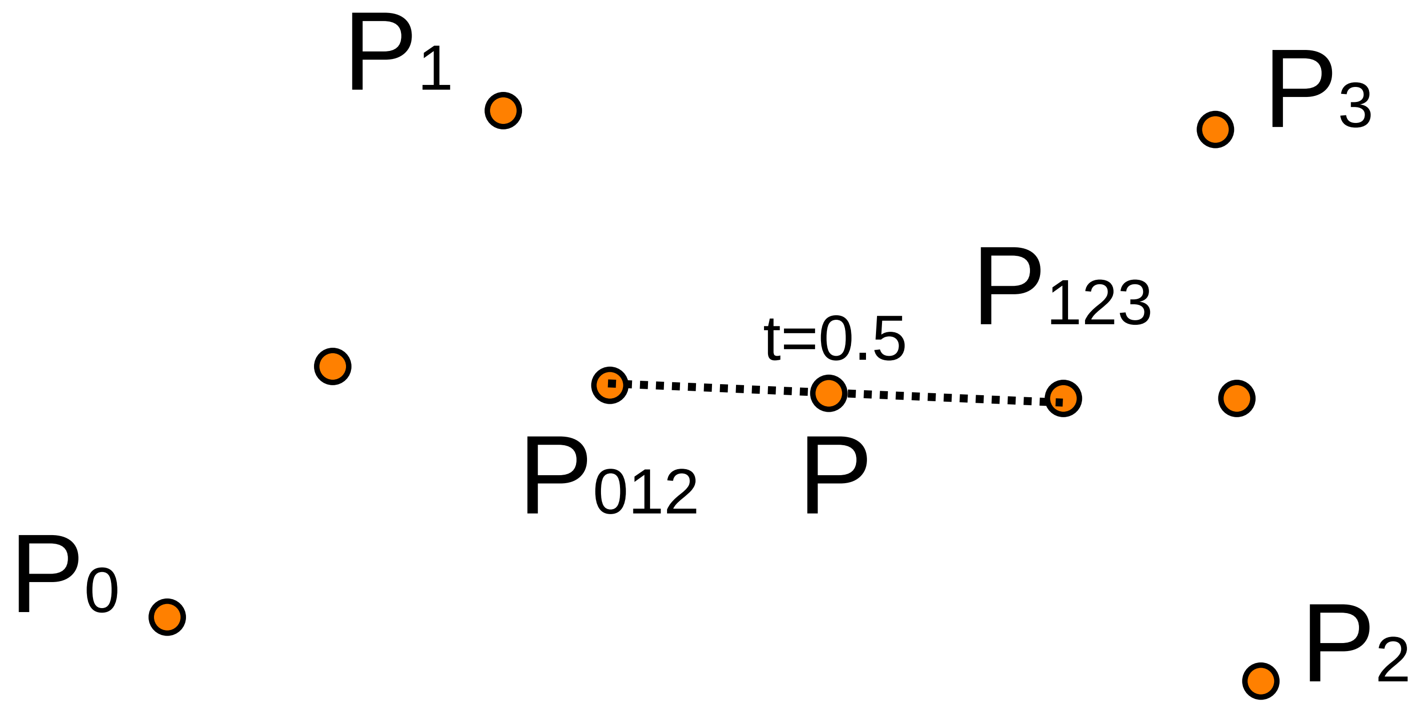
de Casteljau



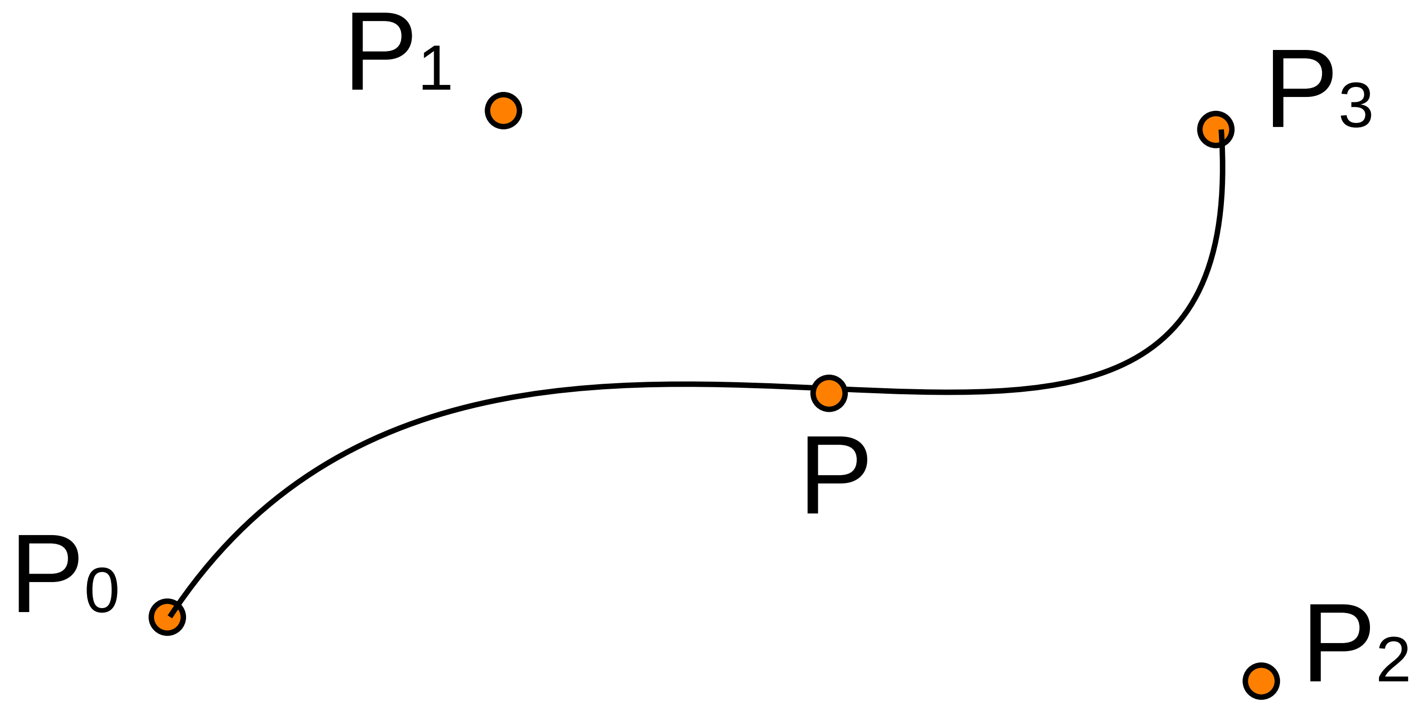
de Casteljau



de Casteljau



de Casteljau



Degree and Order

Linear Interpolation: Degree one curve (m=1), Second Order (2 control points)
Quadratic Interpolation: Degree two curve (m=2), Third Order (3 control points)
Cubic Interpolation: Degree three curve (m=3), Fourth Order (4 control points)
Quartic Interpolation: Degree four curve (m=4), Fifth Order (5 control points)
Etc…

Bézier curves

This family of curves are known as Bézier curves.

They have the general form:

$$P(t) = \sum_{k=0}^{m} B_k^m(t) P_k$$

where m is the degree of the curve and $P_0 \ldots P_m$ are the control points.

Bernstein polynomials

The coefficient functions $B_k^m(t)$ are called Bernstein polynomials. They have the general form:

$$B_k^m(t) = \binom{m}{k} t^k (1-t)^{m-k}$$

where:

$$\binom{m}{k} = \frac{m!}{k!(m-k)!}$$

is the binomial function.

Binomial Function

Remember Pascal's triangle

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}
$$

Bernstein polynomials

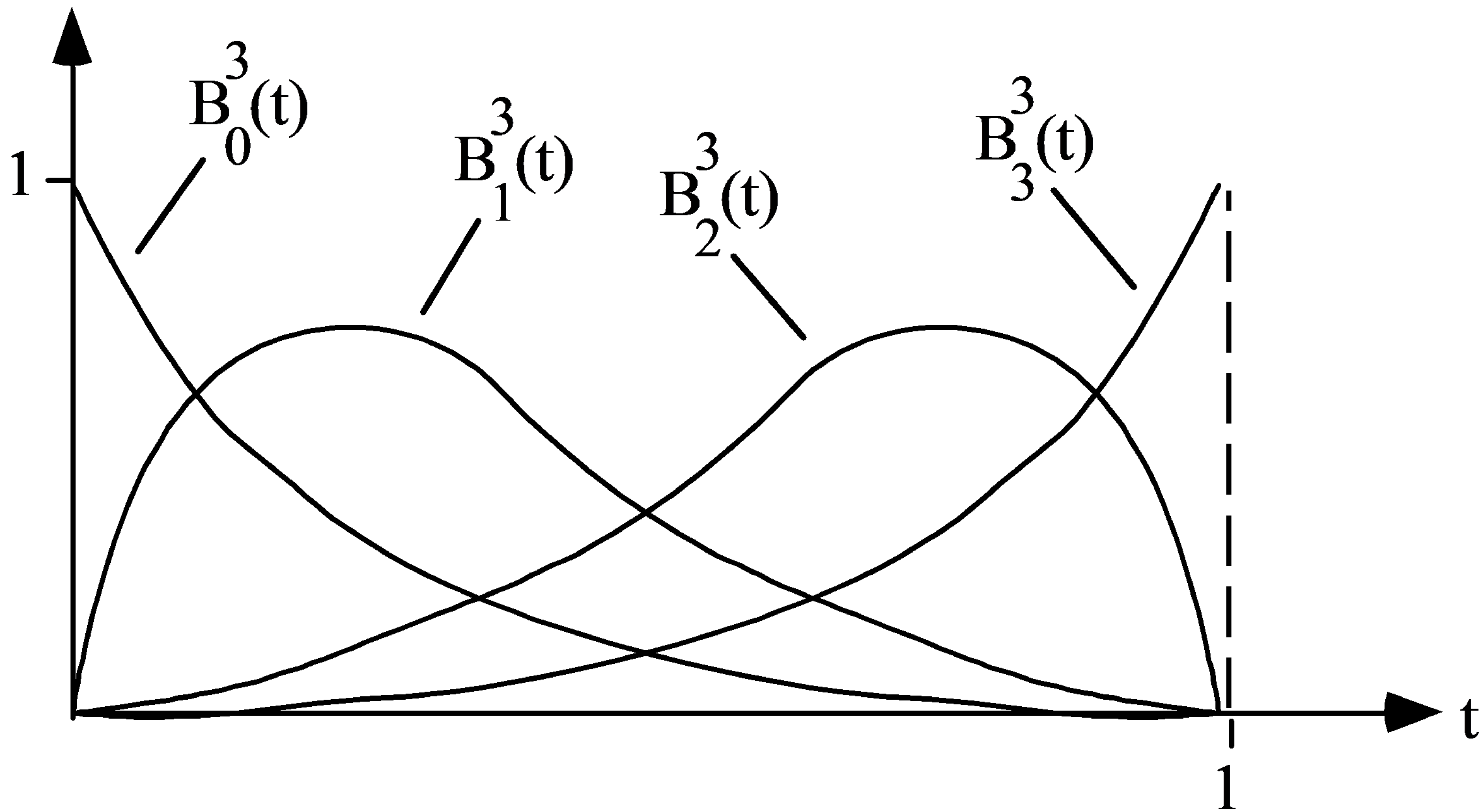
$$B_k^m(t) = \binom{m}{k} t^k (1-t)^{m-k}$$

For the most common case, m = 3:

$$\begin{aligned} B_0^3(t) &= (1-t)^3 \\ B_1^3(t) &= 3t(1-t)^2 \\ B_2^3(t) &= 3t^2(1-t) \\ B_3^3(t) &= t^3 \end{aligned}$$

$$P(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t) P_2 + t^3 P_3$$

Bernstein Polynomials for m = 3



Exercise

$$B_k^m(t) = \binom{m}{k} t^k (1-t)^{m-k}$$

What are the Bernstein polynomials for m = 4?

Solution

$$B_k^m(t) = \binom{m}{k} t^k (1-t)^{m-k}$$

What are the Bernstein polynomials for m = 4?

$$B_0^4\,(\mathrm{t}) = (1-\mathrm{t})^4$$

$$B_1^4\,(\mathrm{t}) = 4\mathrm{t}(1-\mathrm{t})^3$$

$$B_2^4\,(\mathrm{t}) = 6t^2(1-\mathrm{t})^2$$

$$B_3^4\,(\mathrm{t}) = 4t^3(1-t)$$

$$B_3^4\,(\mathrm{t}) = t^4$$

Properties

Bézier curves interpolate their endpoints and approximate all intermediate points.

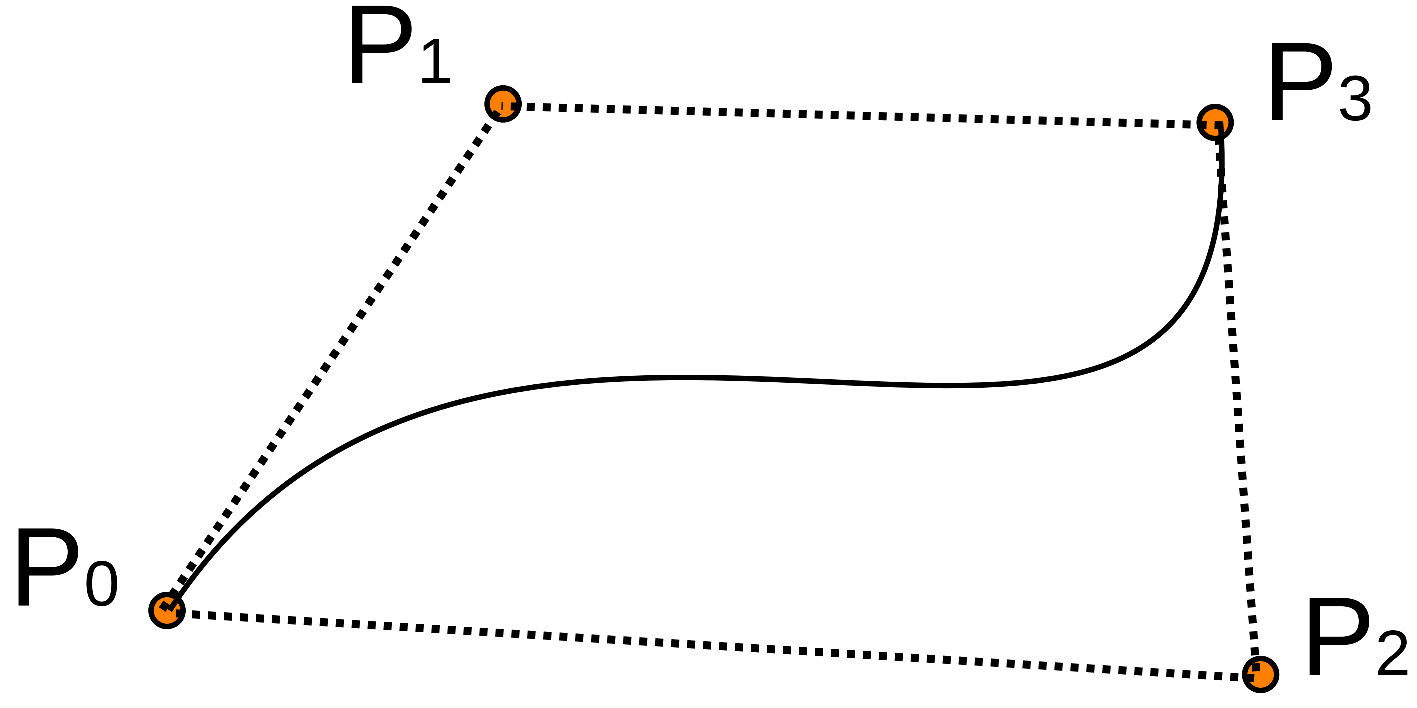
Bézier curves are convex combinations of points:

$$\sum_{k=0}^{m} B_k^m(t) = 1$$

Therefore they are invariant under affine transformation. The transformation of a Bézier curve is the curve based on the transformed control points.

Properties

A Bézier curve lies within the convex hull of its control points:



Tangents

The tangent vector to the curve at parameter t is given by:

$$\begin{aligned}\frac{dP(t)}{dt} &= \sum_{k=0}^{m} \frac{dB_k^m(t)}{dt} P_k \\ &= m \sum_{k=0}^{m-1} B_k^{m-1}(t)(P_{k+1} - P_k)\end{aligned}$$

This is a Bézier curve of degree (m-1) on the vectors between control points.

Exercise

Compute the tangent to at $t = 0.25$ for a quadratic Bezier curve with control points (0,0) (4,4) (8,2)

$$P'(t) = 2 * [(1-t)(P1-P0) + t(P2-P1)]$$

$$P'(0.25) = 2 * [\ (0.75)\ ((4,4) - (0,0)) \ +$$

$$0.25\ ((8,2) - (4,4)\ ]$$

$$= 2 * [\ (0.75)(4,4) + 0.25(4,-2)]$$

$$= 2 * [\ (3,3) + (1, -0.5)] = (8,5)$$

Problem: Polynomial Degree

The degree of the Bernstein polynomials used is coupled to the number of control points: L+1 control points is a combination of L-degree polynomials.

High degree polynomials are expensive to compute and are vulnerable to numerical rounding errors

Problem: Local control

These curves suffer from non-local control.

Moving one control point affects the entire curve.

Each Bernstein polynomial is active (non-zero) over the entire interval (0,1). The curve is a blend of these functions so every control point has an effect on the curve for all t from (0,1)

Splines

A spline is a smooth piecewise-polynomial function (for some measurement of smoothness).

The places where the polynomials join are called knots.

A joined sequence of Bézier curves is an example of a spline.

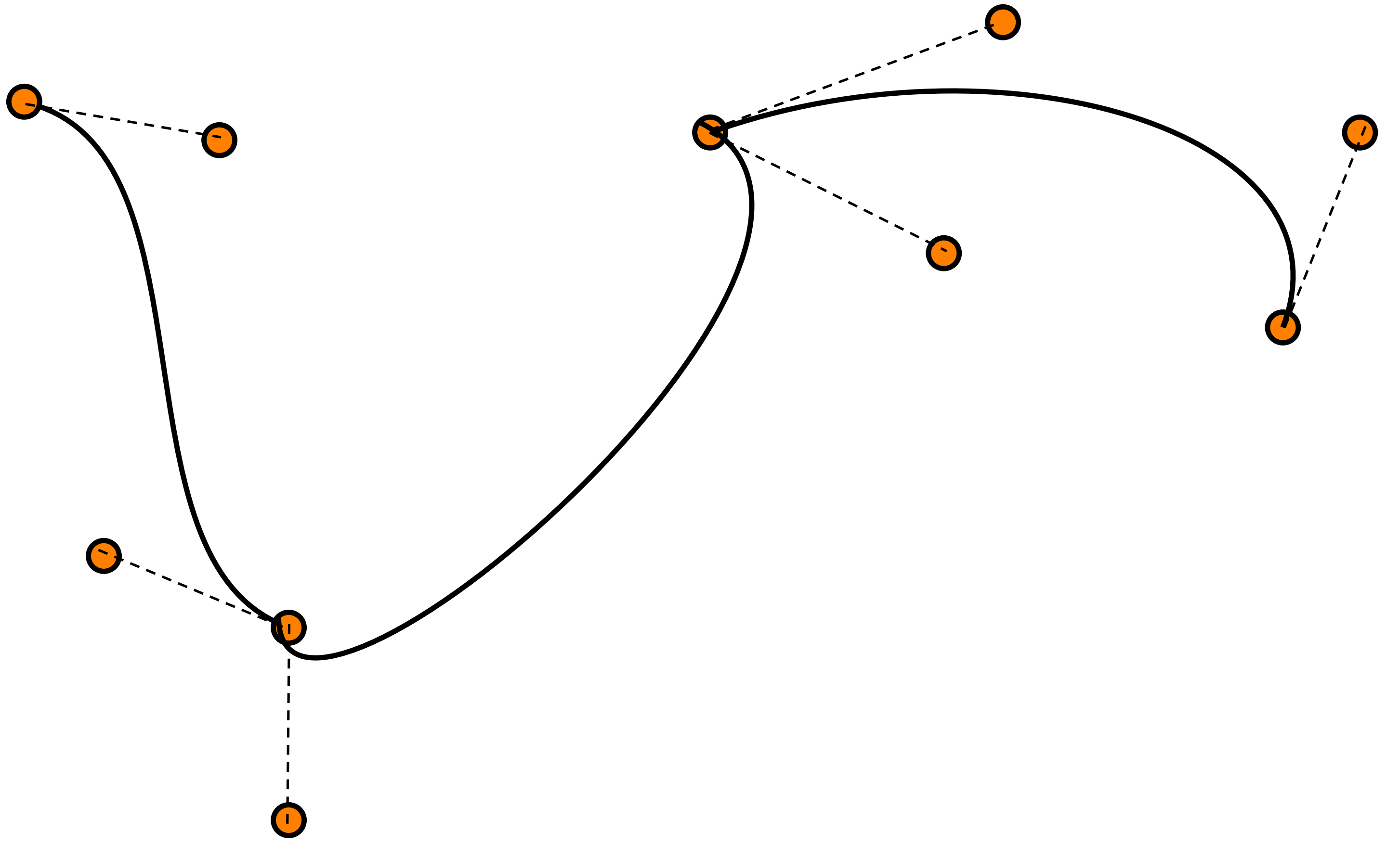
Local control

A spline provides local control.

A control point only affects the curve within a limited neighbourhood.

Bézier splines

We can draw longer curves as sequences of Bézier sections with common endpoints:



3D Modeling

What if we are sick of teapots?

How can we make our own 3d meshes that are not just cubes?

We will look at simple examples along with some clever techniques such as

- Extrusion
- Revolution

Exercise: Cone

How can we model a cone?

There are many ways.

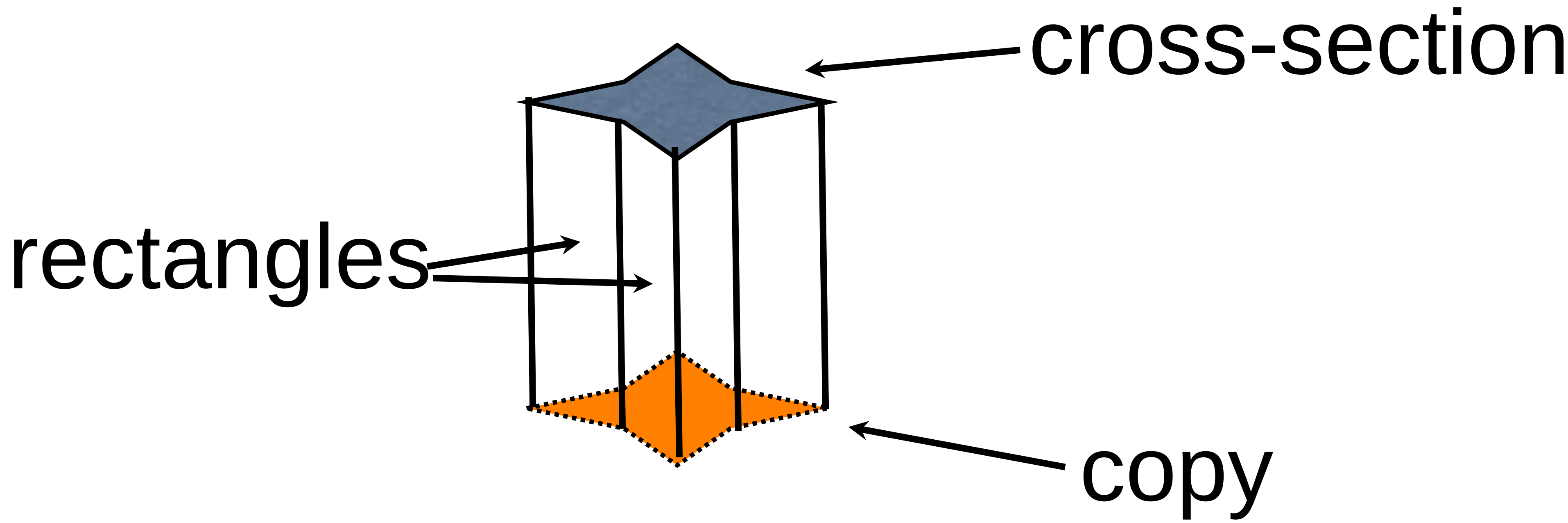
Simple way: Make a circle using a triangle fan parallel to the x-y plane. For example at $z = -3$

Change to middle point to lie at a different z-point for example $z = -1$.

Extruding shapes

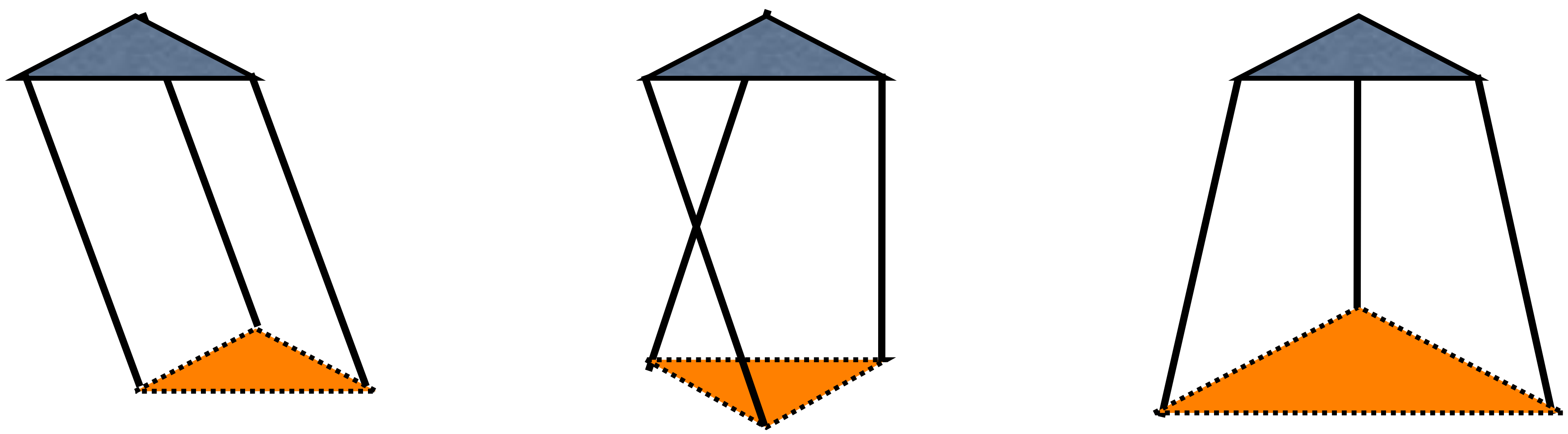
Extruded shapes are created by sweeping a 2D polygon along a line or curve.

The simplest example is a prism.



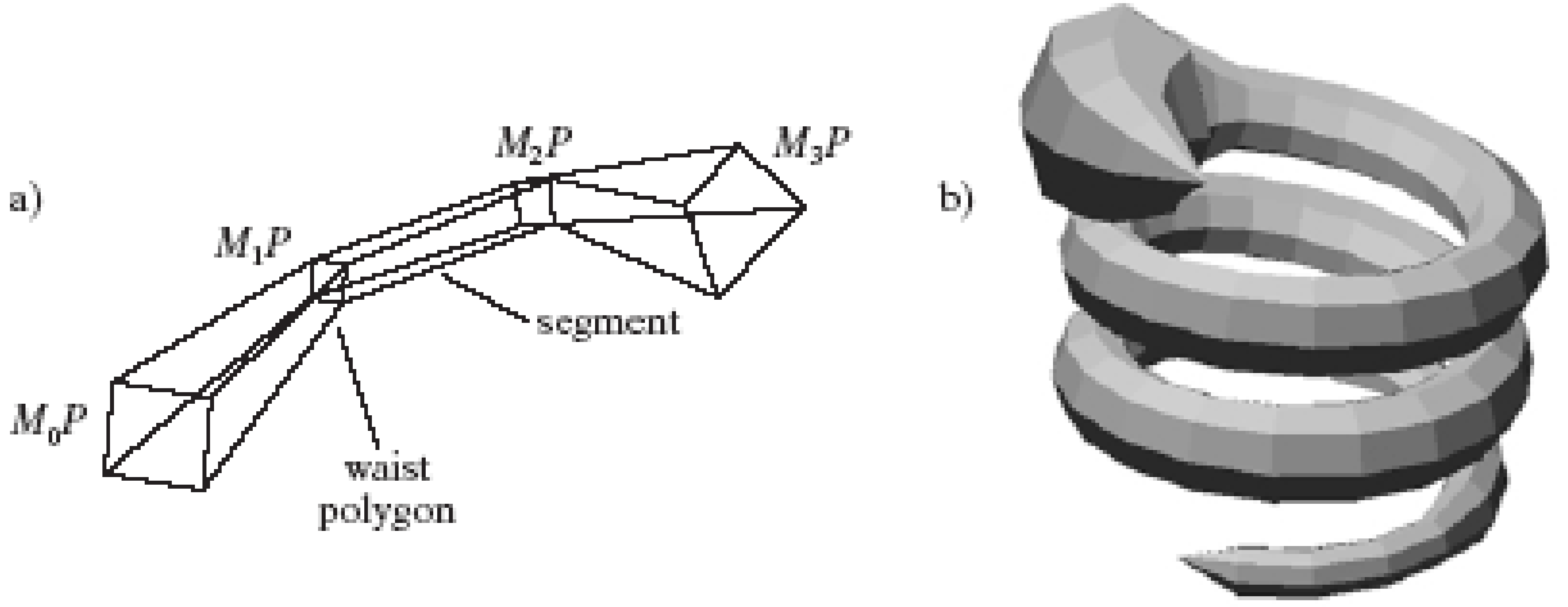
Variations

One copy of the prism can be translated, rotated or scaled from the other.



Segmented Extrusions

A square P extruded three times, in different directions with different tapers and twists. The first segment has end polygons M_0P and M_1P, where the initial matrix M_0 positions and orients the starting end of the tube. The second segment has end polygons M_1P and M_2P, etc.



Segmented extrusions

We can extrude a polygon along a path by specifying it as a series of transformations.

$$\begin{aligned} poly &= P_0, P_1, \ldots, P_k \\ path &= \mathbf{M_0}, \mathbf{M_1}, \ldots, \mathbf{M_n} \end{aligned}$$

At each point in the path we calculate a cross-section:

$$poly_i = \mathbf{M_i} P_0, \mathbf{M_i} P_1, \ldots, \mathbf{M_i} P_k$$

Segmented Extrusion

Sample points along the spine using different values of t

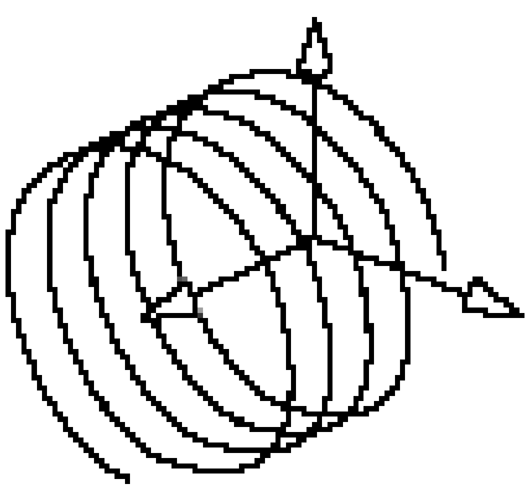
For each t:

- generate the current point on the spine
- generate a transformation matrix
- multiply each point on the cross section by the matrix.
- join these points to the next set of points using quads/triangles.

Segmented Extrusion Example

For example we may wish to extrude a circle cross-section around a helix spine.

helix $C(t) = (cos(t), sin(t), bt))$.



Transformation Matrix

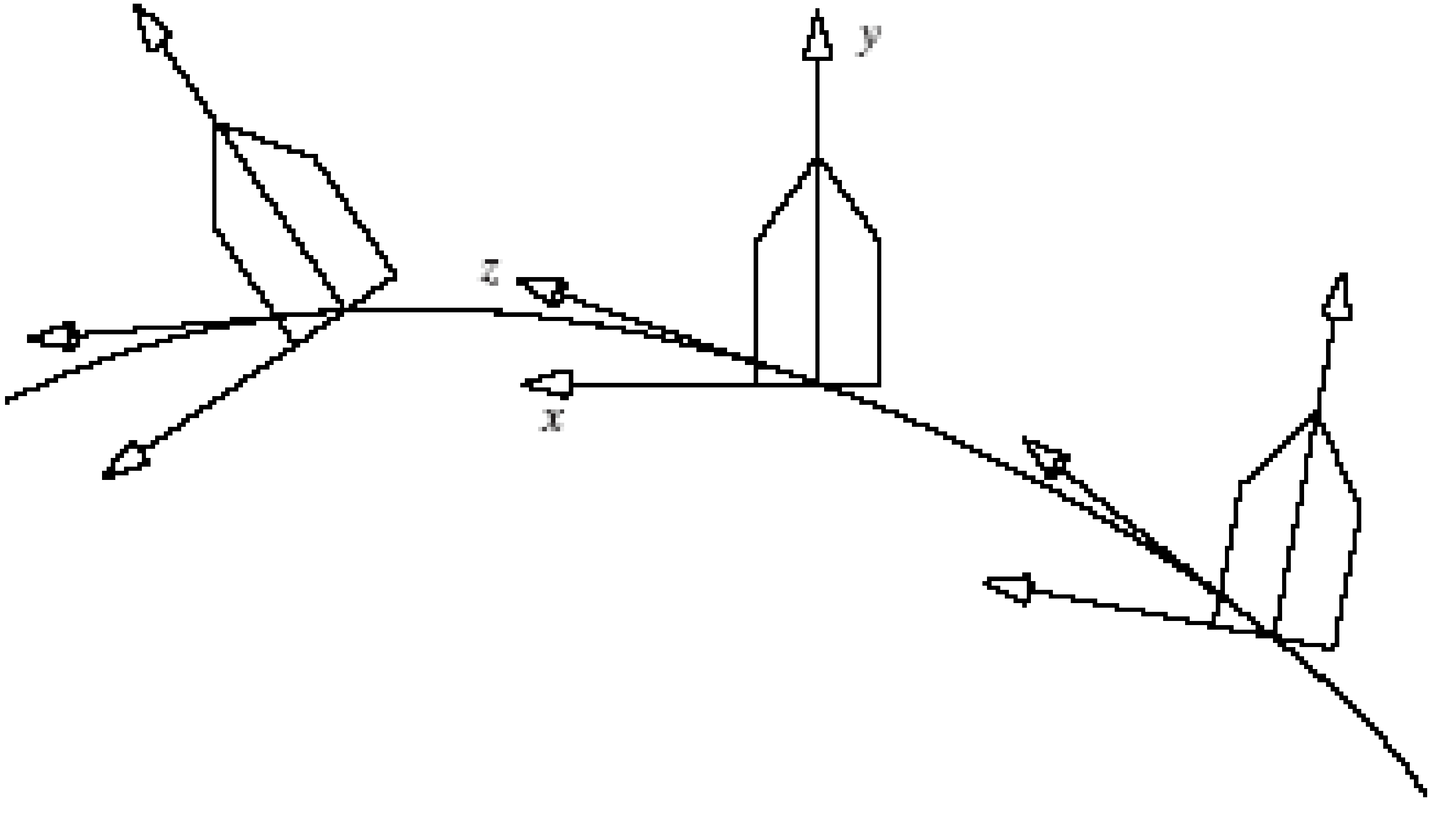
How can we automatically generate a matrix to transform our cross-section by?

We need the origin of the matrix to be the new point on the spine. This will translate our cross-section to the correct location.

Which way will our cross-section be oriented? What should i, j and k of our matrix be?

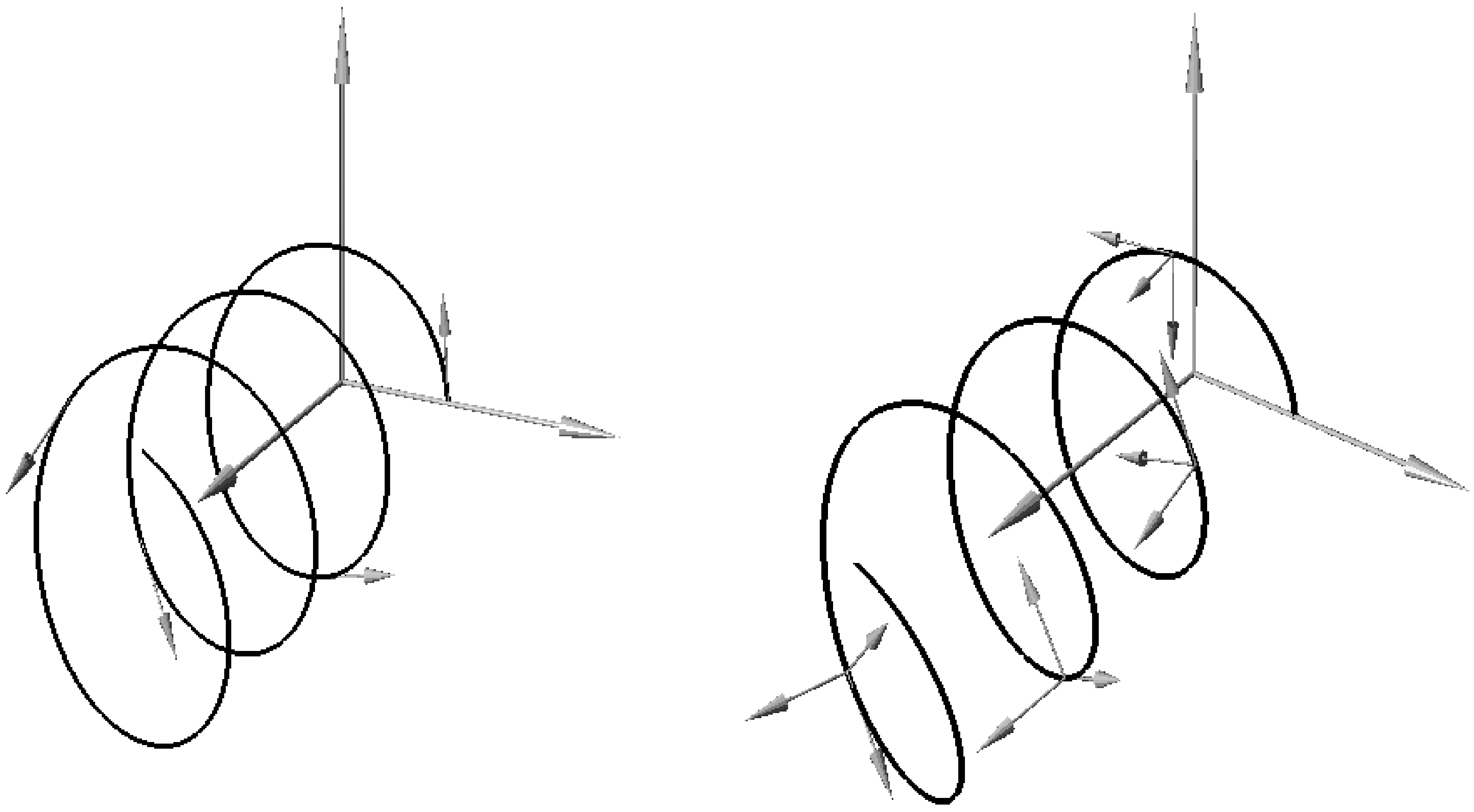
Frenet Frame

We can get the curve values at various points t_i and then build a polygon perpendicular to the curve at $C(t_i)$ using a Frenet frame.



Example

a). Tangents to the helix. b). Frenet frame at various values of t, for the helix.



Frenet Frame

Once we calculate the tangent to the spine at the current point, we can use this to calculate normals.

We then use the tangent and the 2 normals as i, j and k vectors of a co-ordinate frame.

We can then build a matrix from these vectors, using the current point as the origin of the matrix.

Frenet frame

We align the **k** axis with the (normalised) tangent, and choose values of **i** and **j** to be perpendicular.

$$\begin{aligned}
\phi &= \Phi(t) \\
\mathbf{k} &= \hat{\Phi}'(t) \\
\mathbf{i} &= \begin{pmatrix} -k_2 \\ k_1 \\ 0 \end{pmatrix} \\
\mathbf{j} &= \mathbf{k} \times \mathbf{i}
\end{aligned}$$

Frenet Frame Calculation

Finding the tangent (our k vector):

1. Using maths. Eg for

   C(t) = (cos(t), sin(t), bt)

   T(t) = normalise(-sin(t),cos(t),b)

2. Or just approximate the tangent

   T(t) = normalise(C(t+1) – C(t-1))

Frenet Frame Calculation

If our tangent at t is the vector

$T(x,y,z)$

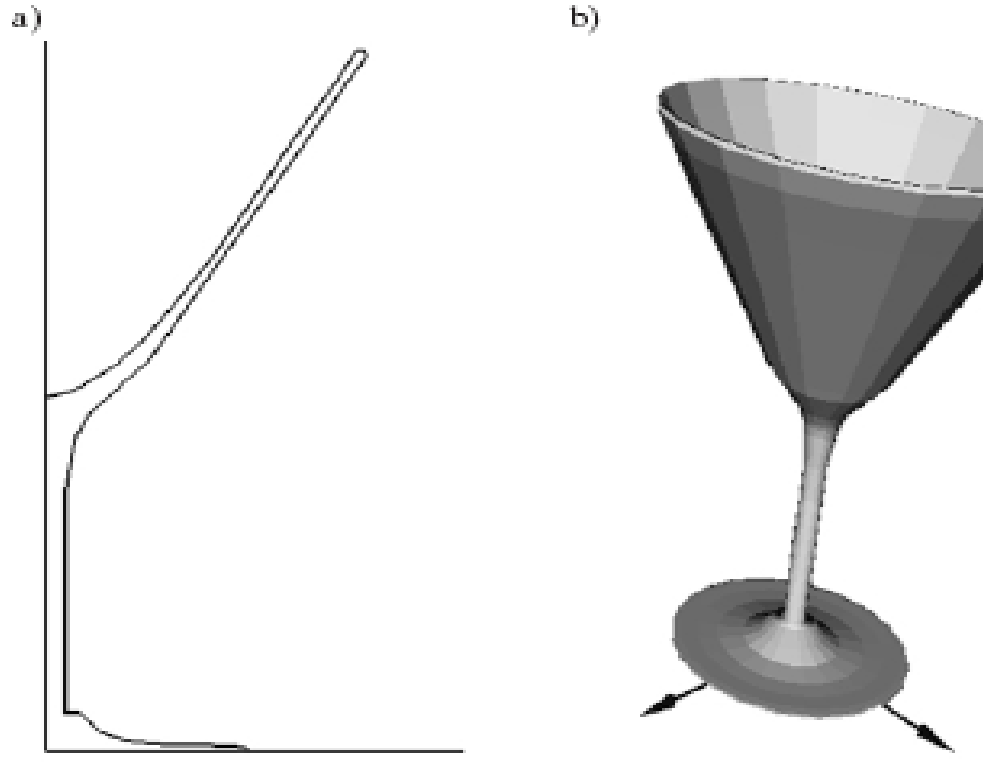
We can use the normal

$N(-y,x,0)$. This will be our i vector

To find the other normal we simply do k x i

Revolution

A surface with radial symmetry (i.e. a round object, like a ring, a vase, a glass) can be made by sweeping a half cross-section around an axis.

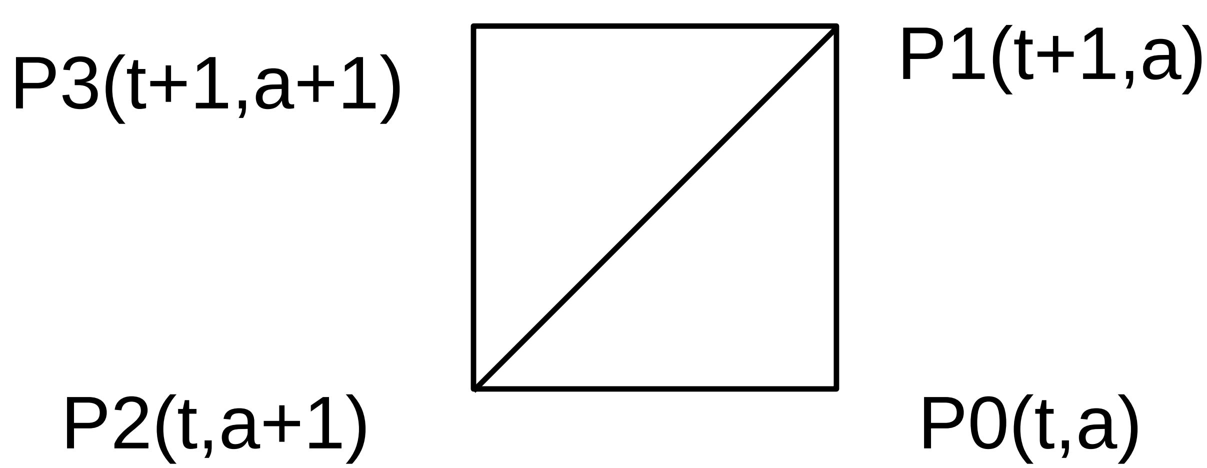


Revolution

Take your 2d function which can generate points for $X(t)$ and $Y(t)$ and sample them for different values of t and angles of a (angle of rotation around axis).

//Revolution around the Y-axis

$$P(t,a) = (X(t)\cos a,\ Y(t),\ X(t)\sin a)$$



L-Systems

A Lindenmayer System (or L-System) is a method for producing fractal structures.

They were initially developed as a tool for modelling plant growth.

http://madflame991.blogspot.com.au/p/lindenmayer-power.html

Rewrite rules

An L-system is a formal grammar:
a set of symbols and rewrite rules. Eg:

Symbols:

A, B, +, -

Rules:

$A \rightarrow B - A - B$

$B \rightarrow A + B + A$

Iteration

We start with a given string of symbols and then iterate, replacing each on the left of a rewrite rule with the string on the right.

A

B - A - B

A + B + A - B - A - B - A + B + A

B - A - B + A + B + A + B - A - B - ...

Drawing

Each string has a graphical interpretation, usually using turtle graphics commands:

A = draw forward 1 step

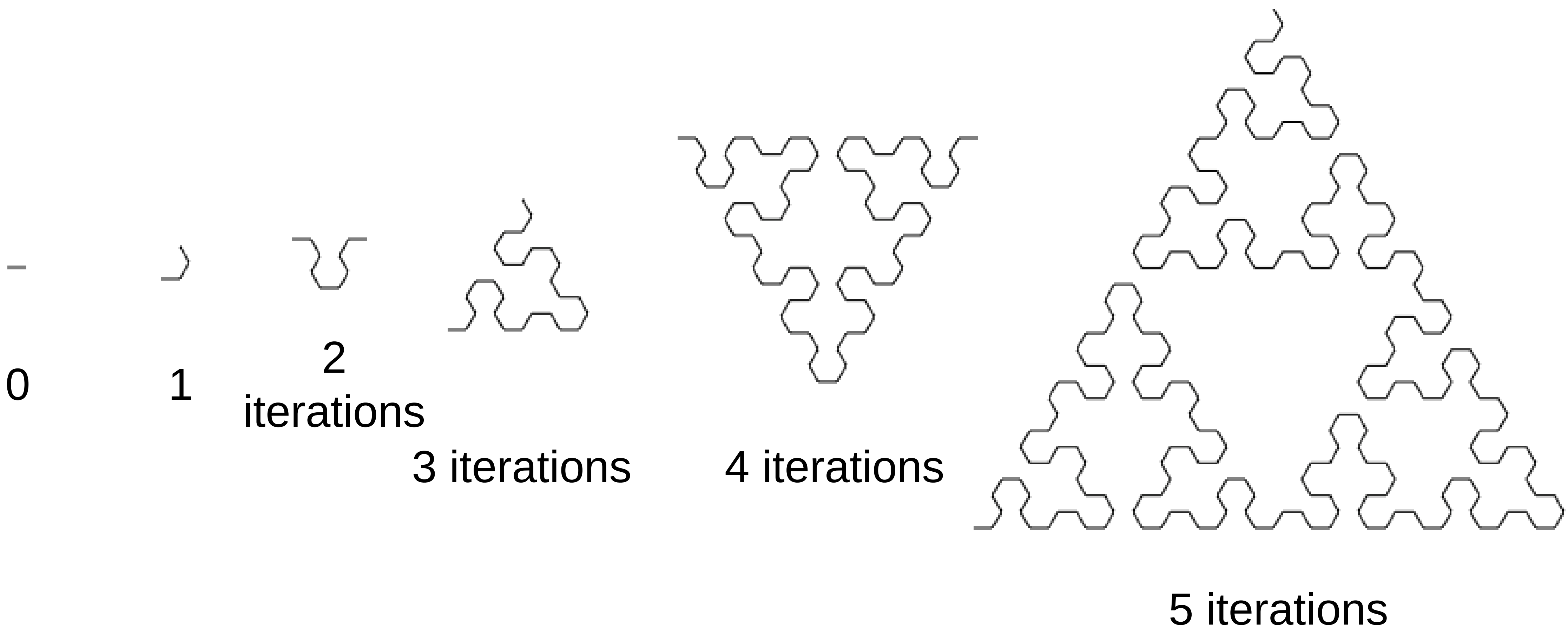
B = draw forward 1 step

+ = turn left 60 degrees

- = turn right 60 degrees

Sierpinski Triangle

This L-System generates the fractal known as the Sierpinski Triangle:



Parameters

We can add parameters to our rewrite rules handle variables like scaling:

$A(s) \rightarrow B(s/2) - A(s/2) - B(s/2)$

$B(s) \rightarrow A(s/2) + B(s/2) + A(s/2)$

A(s) : draw forward s units

B(s) : draw forward s units

Push and Pop

We can also use a LIFO stack to save and restore global state like position and heading:

$A \rightarrow B\ [\ +A\ ]\ -\ A$

$B \rightarrow B\ B$

A : forward 10 B : forward 10

+: rotate 45 left - : rotate 45 right

[: push] : pop ;

Stochastic

We can add multiple productions with weights to allow random selection:

(0.5) $A \rightarrow B\ [\ A\ ]\ A$

(0.5) $A \rightarrow A$

$B \rightarrow B\ B$

Example

(0.5) X → F - [[X] + X] + F [+ F X] - X
(0.5) X → F - F [+ F X] + [[X] + X] - X
F → F F

3D L-Systems

We can build 3D L-Systems by allowing symbols to translate to models and transformations of the coordinate frame.

C : draw cylinder mesh
F : translate(0,0,10)
X : rotate(10, 1, 0, 0)
Y : rotate(10, 0, 1, 0)
S : scale(0.5, 0.5, 0.5)

Example

S -> A [+ B] + A

A -> A - A + A - A

B -> BA

After 1 iteration?

After 2 iterations?

After 3 iterations?

: A forward 10

: + rotate 45 (CW)

: - rotate -90

: [push

:] pop

Example in Format For Web Demo

-> S

1 A [+ B] + A

-> A

1 A - A + A - A

-> B

1 BA

: A

forward 10

: +

rotate 45

: -

rotate -90

: [

push

:]

pop

Example Generation

S -> A [+ B] + A

A -> A - A + A - A

B -> BA

After 1 iteration?

A [+ B] + A

After 2 iterations?

A-A+A-A [+ BA] + A-A+A-A

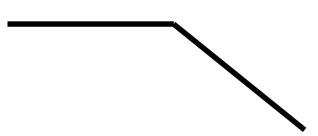
After 3 iterations?

A – A + A – A – A - A + A - A + A - A + A – A ETC

Example Drawing

After 1 iteration?

A [+ B] + A



: A forward 10

: + rotate 45 (CW)

: - rotate -90

: [push

:] pop

Example Drawing

After 2 iterations?

A-A+A-A [+ BA] + A-A+A-A

: A forward 10

: + rotate 45 (CW)

: - rotate -90

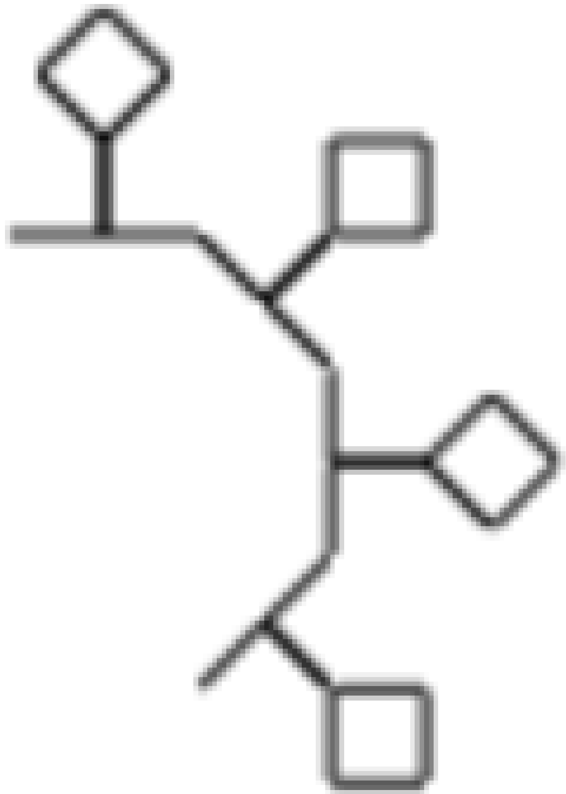
: [push

:] pop

Example Drawing

3 iterations?

A - A + A - A - A - A +
A - A + A - A + A - A
- A - A + A - A [+ BA
] + A - A + A - A - A -
A + A - A + A - A + A -
A - A - A + A - A



Algorithmic Botany

You can read a LOT more here:

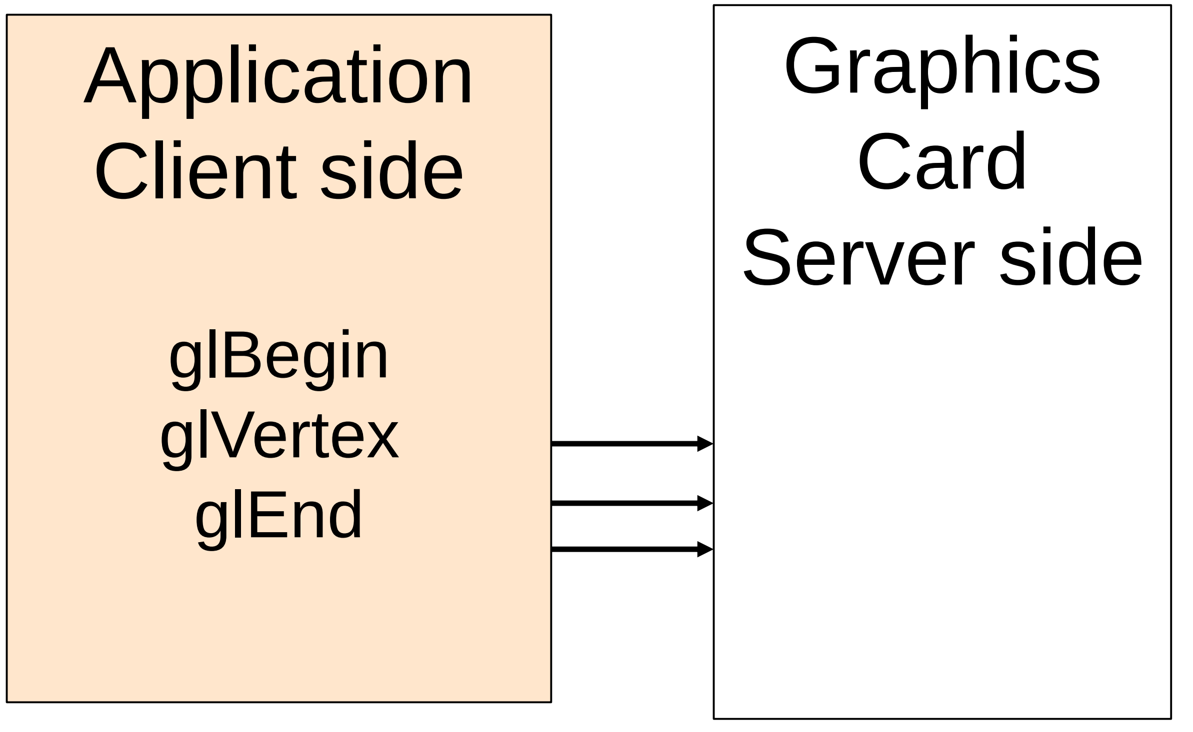
http://algorithmicbotany.org/papers/

Immediate Mode

Primitives are sent to pipeline and displayed right away

More calls to OpenGL commands

No memory of graphical entities on server side – Primitive data lost after drawing which is inefficient if we want to draw object again



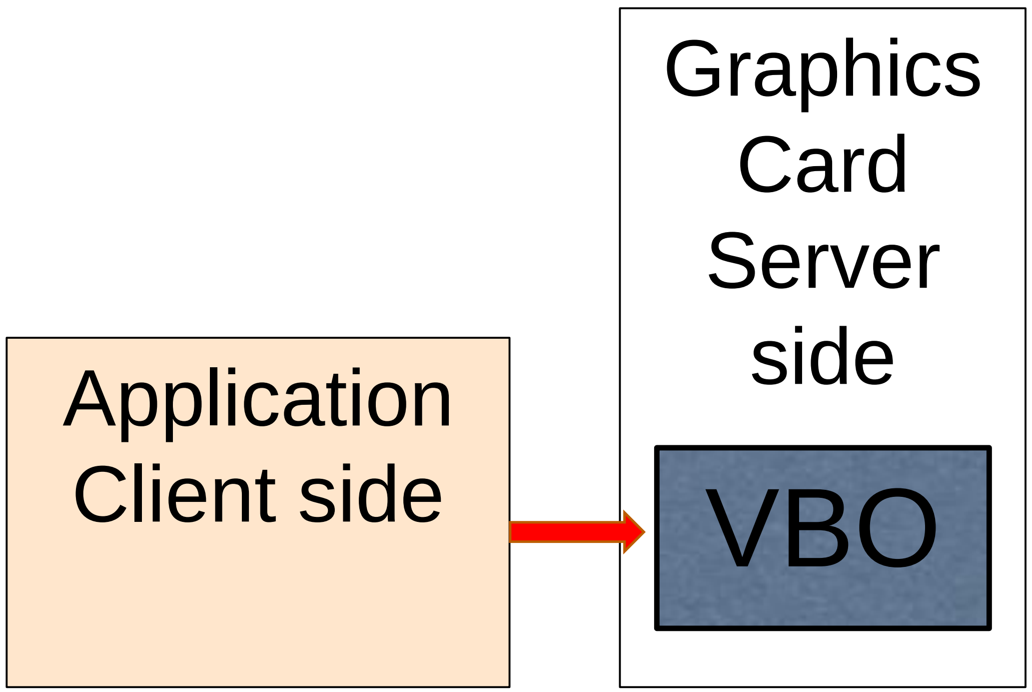
Immediate Mode Example

```
glBegin(GL2.GL_TRIANGLES);{
   gl.glVertex3d(0,2,-4);
   gl.glVertex3d(-2,-2,-4);
   gl.glVertex3d(2,-2,-4);
}gl.glEnd();
```

Retained Mode

Store data in the graphics card's memory instead of retransmitting every time

OpenGL can store data in Vertex Buffer Objects on Graphics Card



Vertices

As we know a vertex is a collection of attributes:

- **position**
- colors
- normal
- etc

VBOs store all this data for all the primitives you want to draw at any one time.

VBOs store this data on the server/graphics card

Client Side Data

```
// Suppose we have 6 vertices with
// positions and corresponding colors in
// our jogl program

float positions[] = {0,1,-1, -1,-1,-1,
                     1,-1,-1, 0, 2,-4,
                     -2,-2,-4, 2,-2,-4};

float colors[] = {1,0,0, 0,1,0,
                  1,1,1, 0,0,0,
                  0,0,1, 1,1,0};
```

Client Side Data

In jogl the VBO commands do not take in arrays.

We need to put them into containers which happen to be called **Buffer**s. These are still client side containers and not on the graphics card memory.

```
FloatBuffer posData =
Buffers.newDirectFloatBuffer(positions);
```

```
FloatBuffer colorData =
Buffers.newDirectFloatBuffer(cols);
```

Our data is now ready to be loaded into a VBO.

Vertex Buffer Objects

VBOs are allocated by glGenBuffers which creates int IDs for each buffer created.

```
//For example generating two bufferIDs

int bufferIDs[] = new int[2];

gl.glGenBuffers(2, bufferIDs,0);
```

VBO Targets

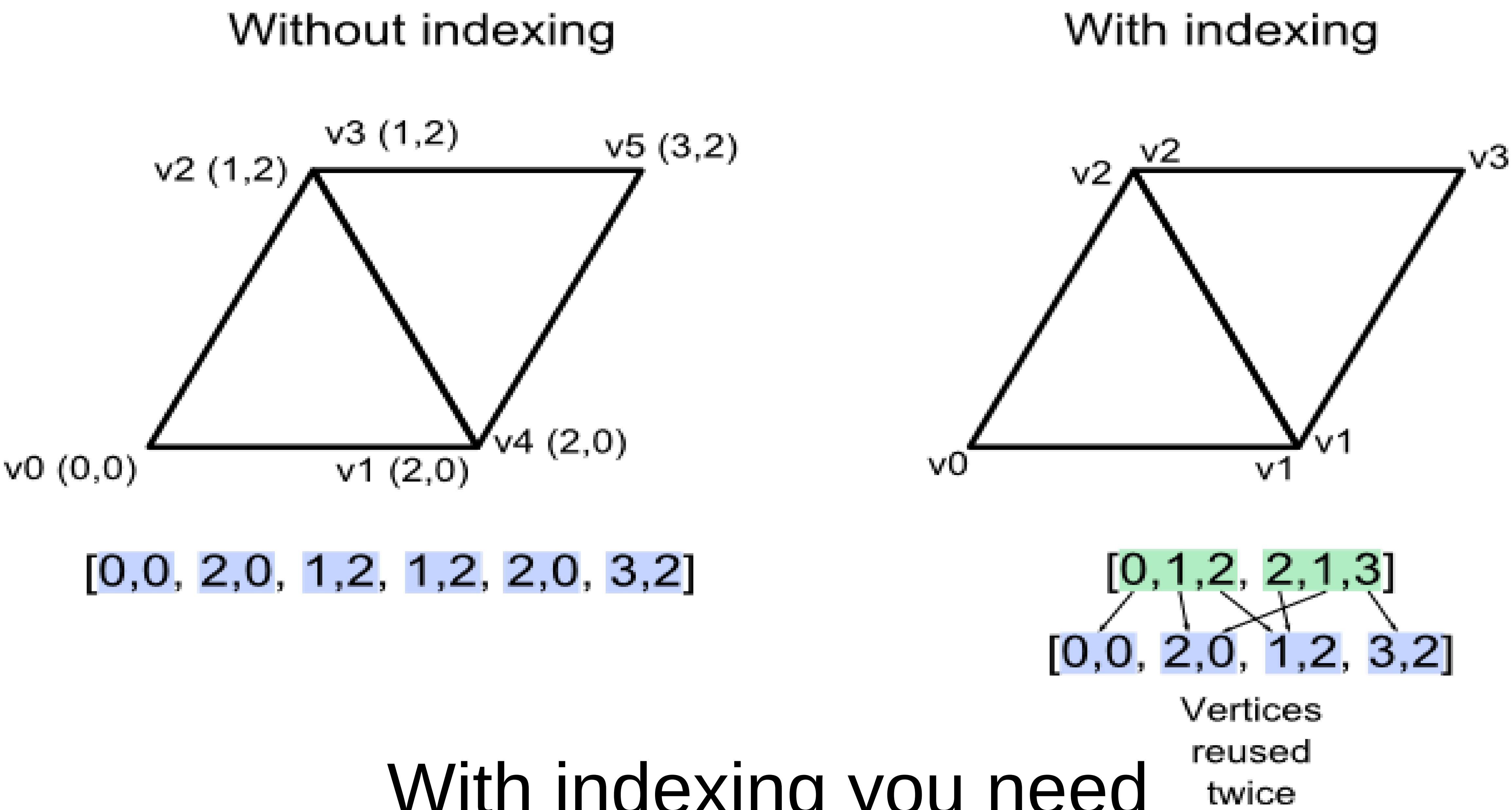
There are different types of buffer objects.

For example:

`GL_ARRAY_BUFFER` is the type used for storing vertex attribute data

`GL_ELEMENT_ARRAY_BUFFER` can be used to store indexes to vertex attribute array data

Indexing



With indexing you need
an extra VBO to store index data.

Binding VBO targets

```
//Bind GL_ARRAY_BUFFER to the
//VBO. This makes the buffer the
//current buffer for
//reading/writing vertex array
//data

gl.glBindBuffer(GL2.GL_ARRAY_BUFFER
, bufferIDs[0]);
```

Vertex Display Buffers

```
// Upload data into the current VBO
gl.glBufferData(int target,
                int size,
                Buffer data,
                int usage);

//target – GL2.GL_ARRAYBUFFER,
//GL2.GL_ELEMENT_ARRAY_BUFFER etc

//size – of data in bytes

//data – the actual data

//usage – a usage hint
```

VBO Usage Hints

GL2.GL_STATIC_DRAW: data is expected to be used many times without modification. Optimal to store on graphics card.

GL2.GL_STREAM_DRAW: data used only a few times. Not so important to store on graphics card

GL2.GL_DYNAMIC_DRAW: data will be changed many times

Vertex Display Buffers

```
// Upload data into the current VBO
// For our example if we were only
// loading positions we could use
gl.glBufferData(GL2.GL_ARRAYBUFFER,
              posData.length*Float.BYTES,
              posData,
              GL2.GL_STATIC_DRAW);
```

Vertex Display Buffers

```
// Upload data into the current VBO
// For our example if we were wanting
// to load position and color data
// we could create an empty buffer of the
// desired size and then load in each
// section of data using glBufferSubData
gl.glBufferData(GL2.GL_ARRAY_BUFFER,
        positions.length*Float.BYTES +
        colors.length*Float.BYTES,
        null, GL2.GL_STATIC_DRAW);
```

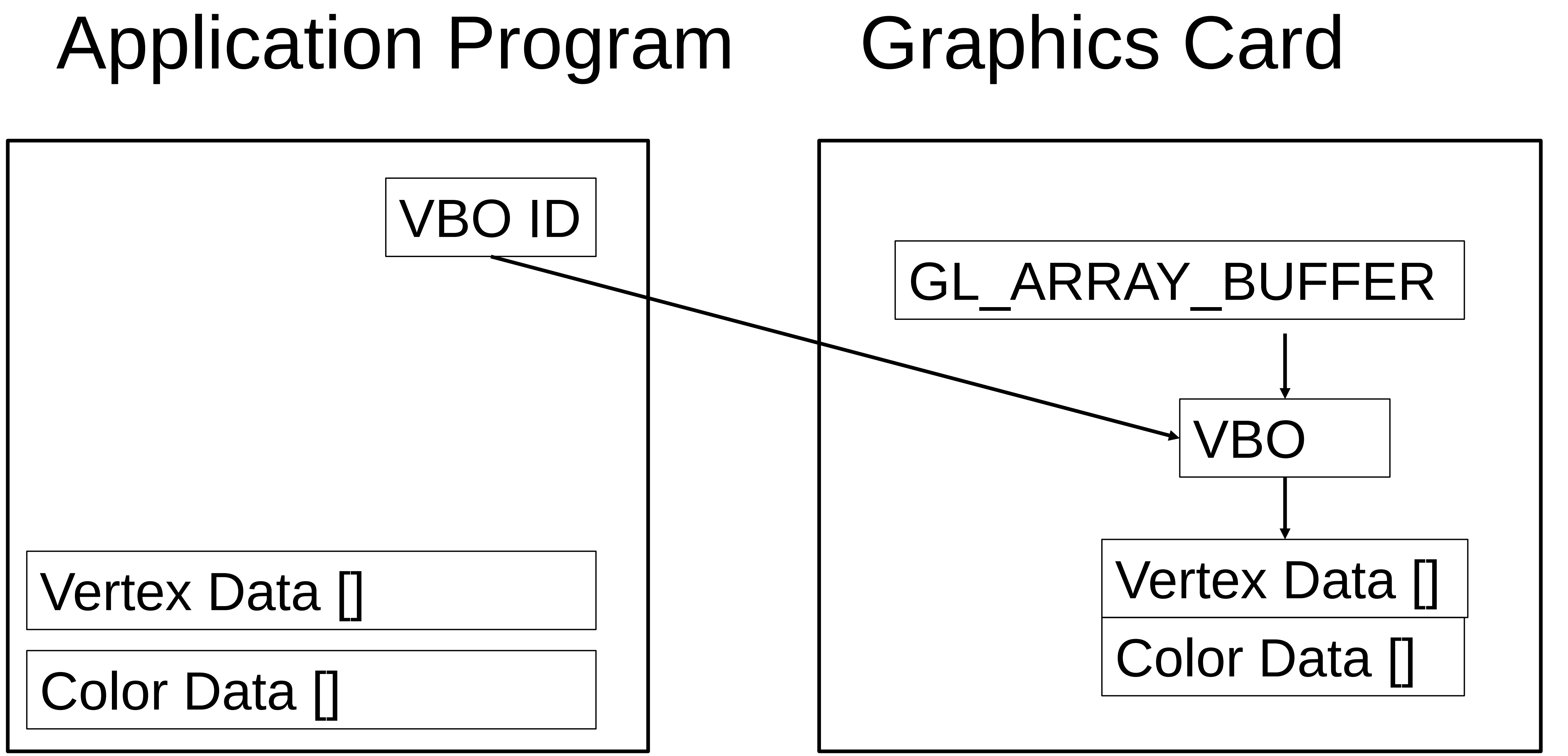
Vertex Display Buffers

```
//Specify part of data stored in the
//current VBO once buffer has been made
//For example vertex positions and color
//data may be stored back to back
gl.glBufferSubData(int target,
                   int offset, //in bytes
                   int size,   //in bytes
                   Buffer data
);
```

Vertex Display Buffers

```
//Specify part of data stored in the
//current VBO once buffer has been made
//For example vertex positions and color
//data may be stored back to back
gl.glBufferSubData(GL2.GL_ARRAY_BUFFER,
0,positions.length*Float.BYTES,posData);

gl.glBufferSubData(GL2.GL_ARRAY_BUFFER,
 positions.length*Float.BYTES, //offset
colors.length*Float.BYTES,colorData);
```

VBOs


Application Program
Graphics Card
VBO ID
GL_ARRAY_BUFFER
VBO
Vertex Data []
Color Data []
Vertex Data []
Color Data []

Using VBOs

All we have done so far is copy data from the client program to the Graphics card. This is done when glBufferData or glBufferSubData is called.

We need to tell the graphics pipeline what is in the buffer – for example which parts of the buffer have the position data vs the color data.

Using VBOs

To tell the graphics pipeline that we want it to use our vertex position and color data

```
//Enable client state

gl.glEnableClientState(
GL2.GL_VERTEX_ARRAY);

gl.glEnableClientState(
GL2.GL_COLOR_ARRAY);

//For other types of data

gl.glEnableClientState(
GL2.GL_NORMAL_ARRAY);//etc
```

Using VBOs with Shaders

To link your vbo to your shader inputs (you get to decide what they are called and used for), instead of gl.glEnableClientState,

```
//assuming the vertex shader has
//in vec4 vertexPos;
int vPos =
gl.glGetAttribLocation(shaderprogram,
"vertexPos");
gl.glEnableVertexAttribArray(vPos);
```

Using VBOs

```
//Tell OpenGL where to find data
gl.glVertexPointer(int size,
                   int type,
                   int stride,
                   long vboOffset);
//size - number of co-ords per vertex
//type - GL2.GL_FLOAT etc
//stride - distance in bytes between
beginning of vertex locations.
//vboOffset - offset in number of bytes
of data location
```

Using VBOs

```
//Tell OpenGL where to find other data.
//Must have 1-1 mapping with the vertex
//array
gl.glColorPointer(int size,
                   int type,
                   int stride,
                   long vboOffset);
gl.glNormalPointer(int type,
                   int stride,
                   long vboOffset);
```

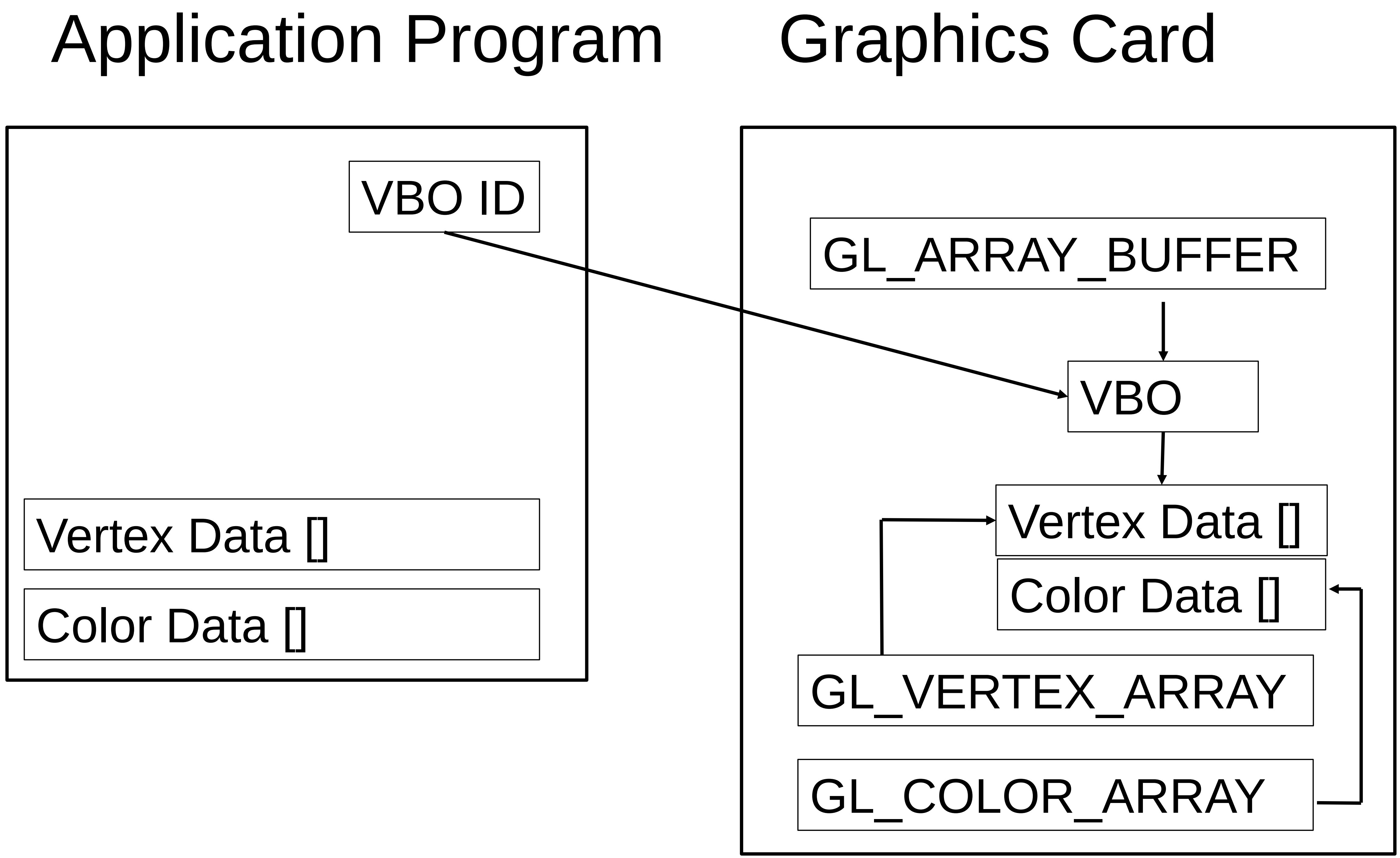
Using VBOs

```
// Tell OpenGL where to find data
// In our example each position has 3
// float co-ordinates. Positions are not
// interleaved with other data and are
// at the start of the buffer
gl.glVertexPointer(3,GL.GL_FLOAT,0, 0);

// In our example color data is found
// after all the position data
gl.glColorPointer(3,GL.GL_FLOAT,0,
positions.length*Float.BYTES );
```

Using VBOs with Shaders

```
//Tell OpenGL where to find data

gl.glVertexAttribPointer(int index,
 int size, int type, boolean normalised,
 int stride, long vboOffset);
//index - shader attribute index
//normalised - whether to normalize the
//data
gl.glVertexAttribPointer(vPos,3,
GL.GL_FLOAT, false,0, 0);
```

VBOs



Application Program
Graphics Card
VBO ID
Vertex Data []
Color Data []
GL_ARRAY_BUFFER
VBO
Vertex Data []
Color Data []
GL_VERTEX_ARRAY
GL_COLOR_ARRAY

Drawing with VBOs

```
// Draw something using the data
// sequentially
gl.glDrawArrays(int mode,
                int first,
                int count);
//mode - GL_TRIANGLES etc
//first - index of first vertex to be
//drawn
//count - number of vertices to be used.
```

Drawing with VBOs

```
//In our example we have data for 2
//triangles, so 6 vertices
//and we are starting at the
//vertex at index 0
gl.glDrawArrays(GL2.GL_TRIANGLES,0,6);

//This would just draw the second triangle
gl.glDrawArrays(GL2.GL_TRIANGLES,3,6);
```

Indexed Drawing

```
// Draw something using indexed data
gl.glDrawElements(int mode, int count,
                  int type, long offset);

//mode - GL_TRIANGLES etc
//count - number of indices to be used.
//type - type of index array - should be
//unsigned and smallest type possible.
//offset - in bytes!
```

Indexed Drawing

```
//Suppose we want to use indexes to
//access our data
short indexes[] = {0,1,5,3,4,2};
ShortBuffer indexedData =
Buffers.newDirectShortBuffer(indexes);

//Set up another buffer for //the
indexes
gl.glBindBuffer(GL2.GL_ELEMENT_ARRAY_BUF
FER, bufferIDs[1]);
```

Indexed Drawing

```
//load index data
gl.glBufferData(
            GL2.GL_ELEMENT_ARRAY_BUFFER,
            indexes.length *Short.BYTES,
            indexData,
            GL2.GL_STATIC_DRAW);
 //draw the data
gl.glDrawElements(GL2.GL_TRIANGLES, 6,
GL2.GL_UNSIGNED_SHORT, 0);
```

Updating a VBO

- Copy new data into the bound VBO with

`gl.glBufferData()` or

`glBufferSubData()`

- map the buffer object into client's memory, so that client can update data with the pointer to the mapped buffer

`glMapBuffer()`

Drawing Multiple Objects

Must make many calls each time we draw each new shape.

glBindBuffer

glVertexPointer

glColorPointer

glNormalPointer

etc

Vertex Array Object (VAO)

Encapsulates vbo and vertex attribute states to rapidly switch between states using one openGL call.

```
gl.glBindVertexArray(vaoIDs[0]);
```

First generate vao ids.

```
int vaoIDs[] = new int[2];

gl.glGenVertexArrays(2, vaoIDs,0);
```

Set up VAOs

```
//Assume vbos and vao ids have been set up.
gl.glBindVertexArray(vaoIds[0]);
gl.glBindBuffer(GL2.GL_ARRAY_BUFFER,vboIds[0]);
gl.glEnableClientState…
gl.glVertexPointer… //etc other calls
gl.glBindVertexArray(vaoIds[1]);
gl.glBindBuffer(GL2.GL_ARRAY_BUFFER,vboIds[1]);
gl.glEnableClientState..
gl.glVertexPointer
//etc other calls
```

VAO switching

```
//Once vaos have been set up
gl.glBindVertexArray(vaoIds[0]);
gl.glDrawArrays(GL2.GL_TRIANGLES,0,N);
gl.glBindVertexArray(vaoIds[1]);
gl.glDrawArrays(GL2.GL_TRIANGLES,0,M);
//Use no vao
gl.glBindVertexArray(0);
```

Deleting a VBOs and VAOs

To free VBOs and VAOs once you do not need them anymore.

```
gl.glDeleteBuffers(2,vboIds,0);
gl.glDeleteVertexArray(2,vaoIds,0);
```