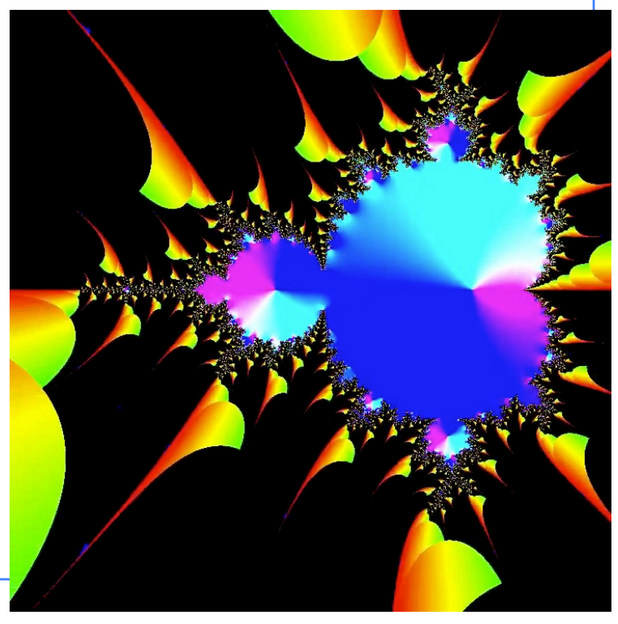
TSBK07/TSBK11

Computer Graphics

Ingemar Ragnemalm, ISY



Lecture 10

Mid-course (sort of) evaluation/feedback

Object representation

Splines

Animation

Online interactive feedback

https://ragnemalm.se/quiz

Labs and duggas

1) Lab progress is sometimes slow. We want to help you.

Stuck with a bug for a long time? Let me know.

2) I am evaluating the dugga system for future revisions.
(But I can't change it mid-course.)

Lecture questions

1. How can you procedurally build a sphere?

2. What is the sum of a set of blending functions?

3. What is the difference between C^1 and G^1?

4. Why does the Bézier stay in the convex hull of the control points?

5. Why are Catmull-Rom splines good for animations?

Instancing

Draw a large number of the same model!

Each instance has an index, the instance number.

glDrawArraysInstanced(GL_TRIANGLES, 0, 3, 10);

draws a triangle 10 times!

gl_InstanceID tells the shader which instance we have.
Use for affecting position.

Billboard instancing demo

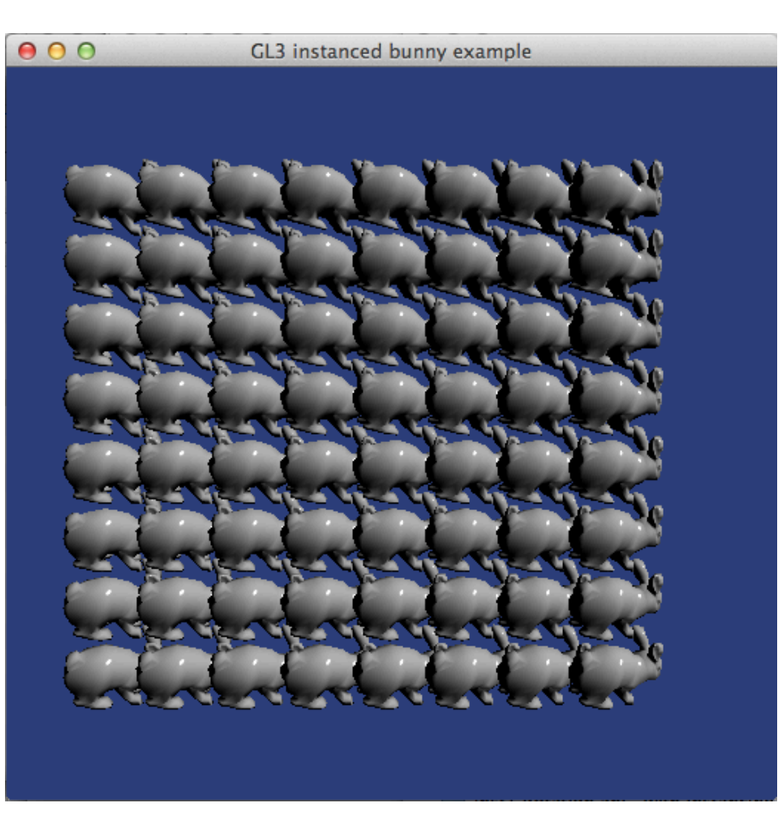
One single call to glDrawArraysInstanced

Position trivially affected by gl_Instance ID

```
#version 150

in  vec3 in_Position;
uniform mat4 myMatrix;
uniform float angle;
uniform float slope;
out vec2 texCoord;

void main(void)
{
	mat4 r;
	float a = angle + gl_InstanceID * 0.5;
	float rr = 1.0 - slope * gl_InstanceID * 0.01;
	r[0] = rr*vec4(cos(a), -sin(a), 0, 0);
	r[1] = rr*vec4(sin(a), cos(a), 0, 0);
	r[2] = vec4(0, 0, 1, 0);
	r[3] = vec4(0, 0, 0, 1);
	texCoord.s = in_Position.x+0.5;
	texCoord.t = in_Position.y+0.5;
	gl_Position = r * myMatrix *  vec4(in_Position,
1.0);
}
```

Instancing complex models

Less significant; A more complex model puts enough load on the system to hide the impact of instancing.



Large worlds, conclusions

High-level VSD to limit processing to visible parts - start with frustum culling

Level-of-detail to reduce unneccessary processing of detailed models

Use billboards for extreme simplification on large distance, particle effects etc

If we can’t see the difference, use the cheaper solution!

3D object representation

In order of importance:

- Polyhedra
- Bi-cubic parametric patches
- Procedural representation, fractals
- Constructive solid geometry
- Implicit representation by quadrics

plus...

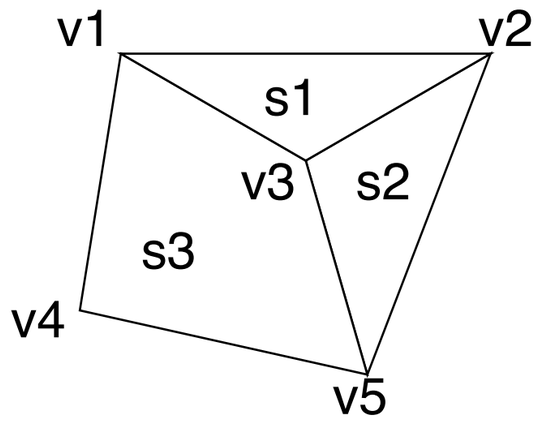
3D object representation

Advanced representations:

- Density volumes/voxel representations
- Point-based representations, including blobby objects
- Level sets

Polyhedra representations

Dominant in real-time graphics
Less suited for off-line rendering



Vertex table

v1 = x1, y1, z1
v2 = x2, y2, z2
v3 = x3, y3, z3
v4 = x4, y4, z4
v5 = x5, y5, z5

Surface table

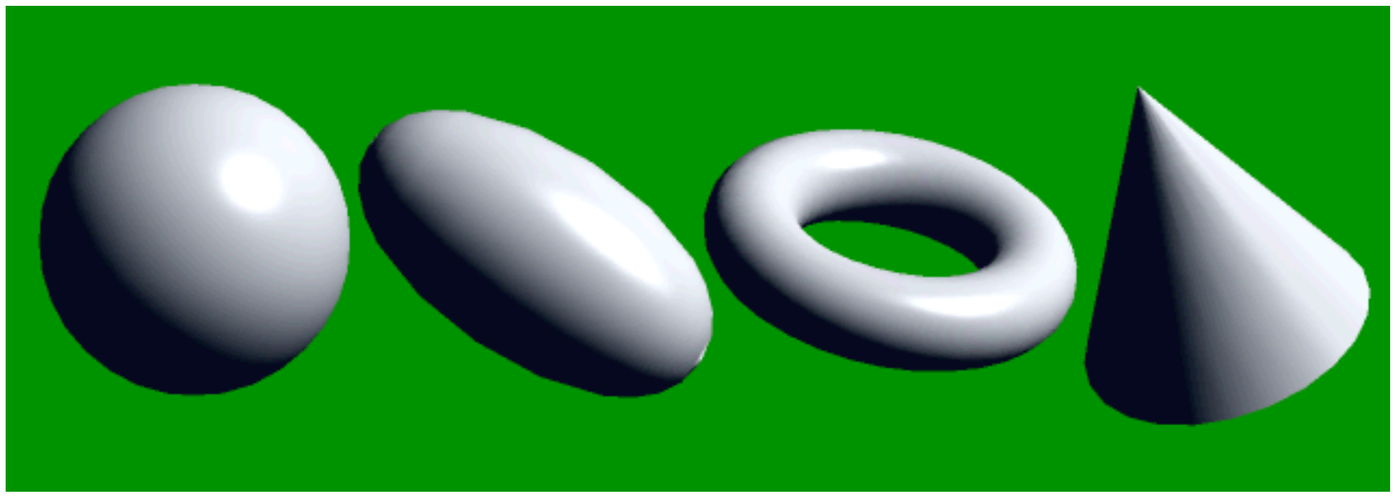
s1 = v1, v2, v3
s2 = v2, v5, v3
s3 = v1, v3, v5, v4

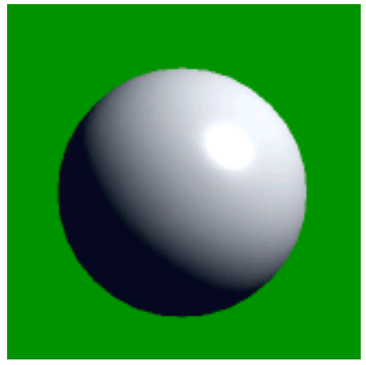
Implicit representations:

Quadric surfaces

Surfaces represented by second-degree polynomials

Sphere Ellipsoid Torus Cone





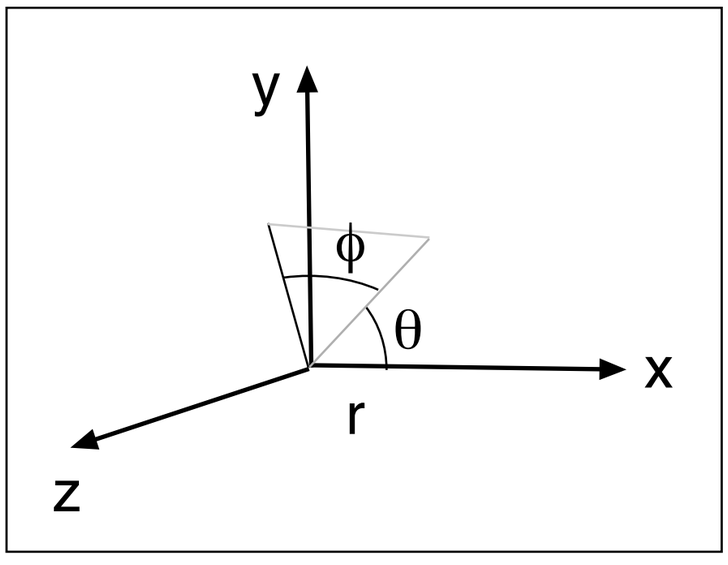
Sphere

Equation:

$$x^2 + y^2 + z^2 = r^2$$

Parametric:

$$x = r \cos \phi \cos \theta$$
$$y = r \cos \phi \sin \theta$$
$$z = r \sin \phi$$



$$-\pi/2 \leq \phi \leq \pi/2$$
$$-\pi \leq \theta \leq \pi$$

Where do we find quadrics?

- Raytracing (especially simpler ones)
- Bounding shapes for VSD
- Simplified collisions
- Basic shapes in 3D modelling programs
- Procedurally modelled to polyhedra

Procedural generation of shapes

Generate geometry from code

Example: Create a sphere

Three different approaches!

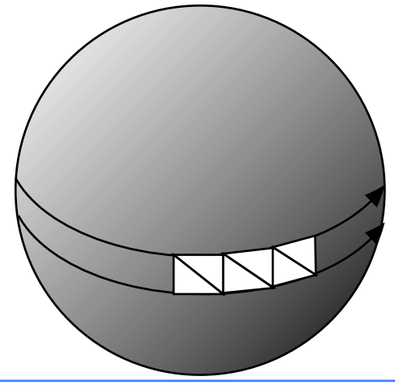
1. Sphere layer by layer

Layer by layer:

Walk along edges using polar coordinates

Typically two layers at the same time

Generate polygons between the layers

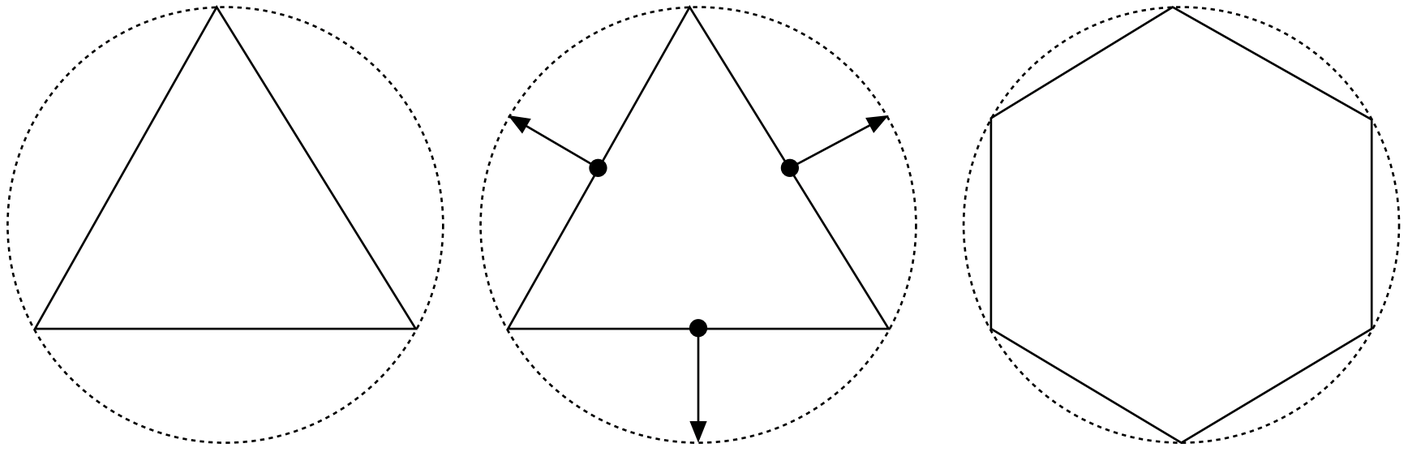


2. Sphere by tesselation

Make base shape (tetrahedron, icosahedron)

Subdivide repeatedly to desired resolution

Keep distance to center point/axis (quadric function)

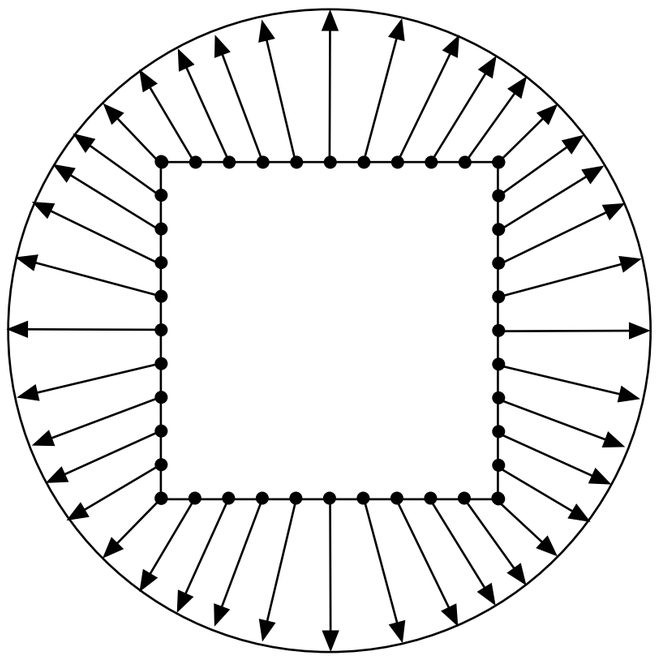


Important case:
Procedural planets.
Start with a cube

3. Map from different shape

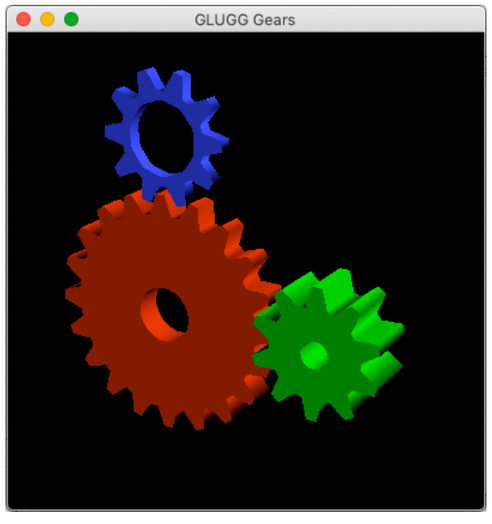
Make base *detailed* shape

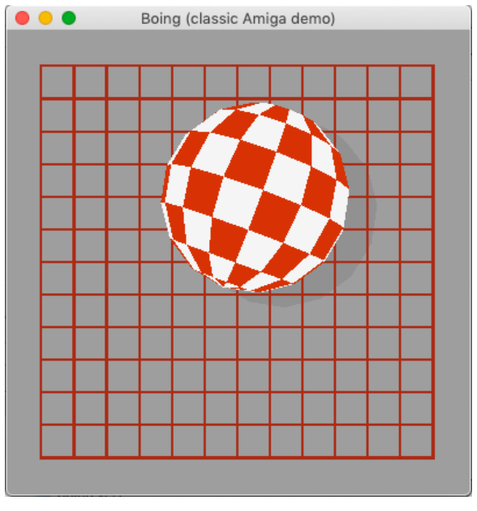
Translate vertices to quadric (typically sphere)

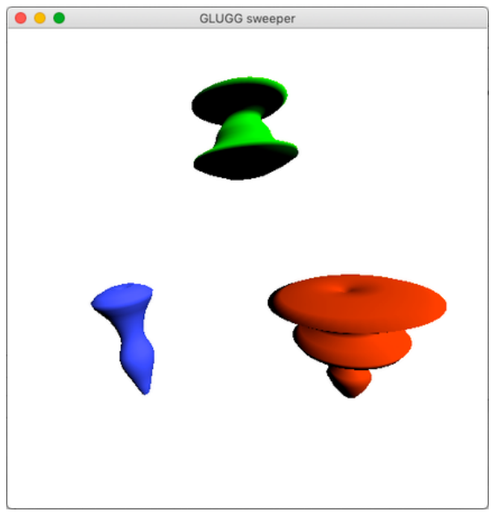


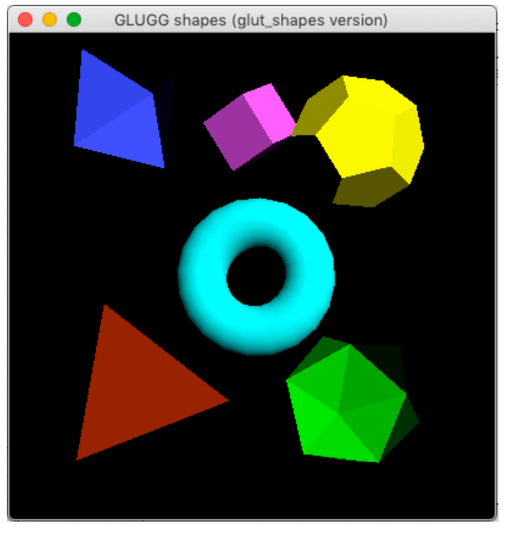
OpenGL Utilities for Geometry Generation (GLUGG)

Creates geometry by passing vertex by vertex and uploads to VAO.
Intended for generating shapes at initialization, as pre-processing.







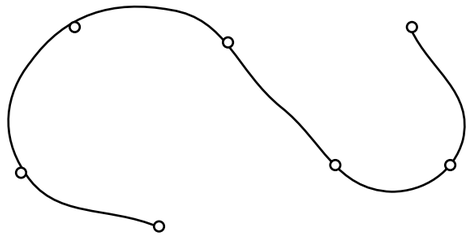


Tesselation of sphere using GLUGG

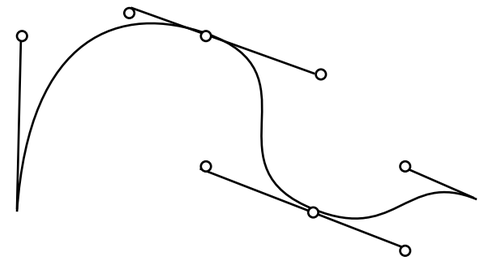
```
vec3 v[]=
{
      {0.0, 0.0, 1.0}, {0.0, 0.942, -0.333},
      {-0.816, -0.471, -0.333},
      {0.816, -0.471, -0.333}
};

static GLfloat theta[] = {0.0,0.0,0.0};

void triangle( vec3 a, vec3 b, vec3 c)
{
      gluggNormalv(a);
      gluggVertexv(a);
      gluggNormalv(b);
      gluggVertexv(b);
      gluggNormalv(c);
      gluggVertexv(c);
}
```

```
void divide_triangle(vec3 a, vec3 b, vec3 c, int m)
{
   vec3 v1, v2, v3;
   int j;
   if(m>0)
   {
      v1 = a + b;
      v1 = normalize(v1);
      v2 = a + c;
      v2 = normalize(v2);
      v3 = b + c;
      v3 = normalize(v3);
      divide_triangle(a, v1, v2, m-1);
      divide_triangle(c, v2, v3, m-1);
      divide_triangle(b, v3, v1, m-1);
      divide_triangle(v1, v3, v2, m-1);
   }
   else
/* draw triangle at end of recursion */
     (triangle(a,b,c));
}
```

```
GLuint tetrahedron( int m, int *count)
{
/* Apply triangle subdivision
to faces of tetrahedron */
      gluggBegin(GLUGG_TRIANGLES);
      divide_triangle(v[0], v[1], v[2], m);
      divide_triangle(v[3], v[2], v[1], m);
      divide_triangle(v[0], v[3], v[1], m);
      divide_triangle(v[0], v[2], v[3], m);
      gm = gluggEndDisposable(count, program, 0);
      return gm.vertexArrayObjID;
}
```



Splines



Originally a drafting tool to create a smooth curve

In computer graphics: a curve built from sections, each described by a 2nd or 3rd degree polynomial.

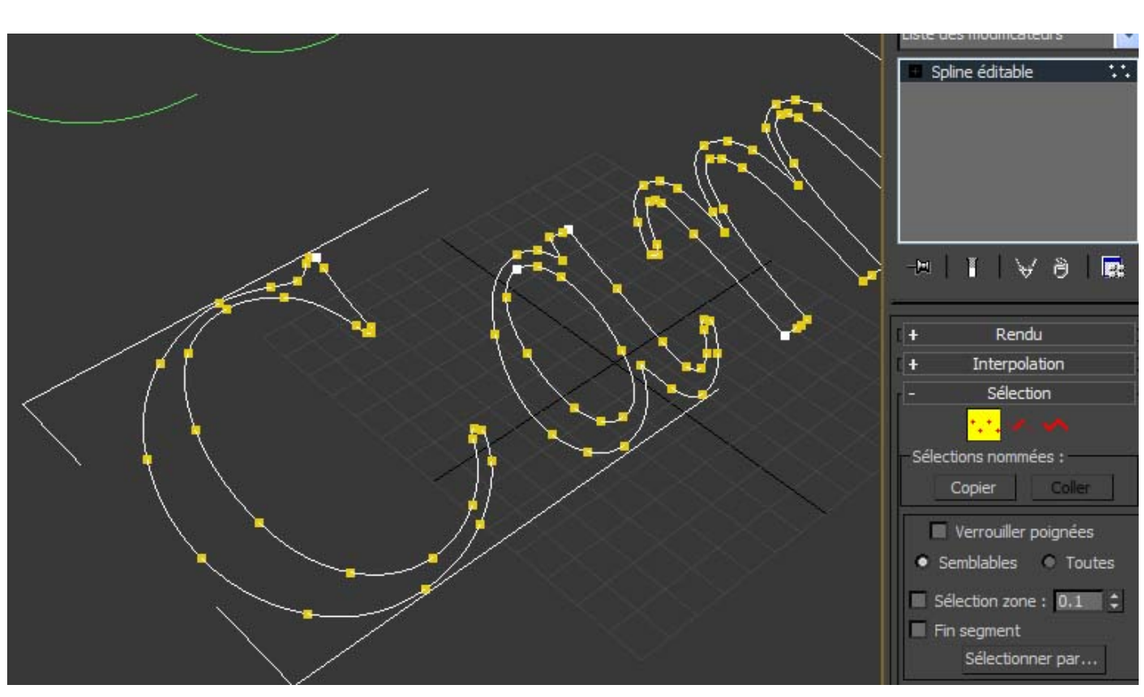
Very common in non-real-time graphics, both 2D and 3D!

Useful also for real-time.

Applications of splines

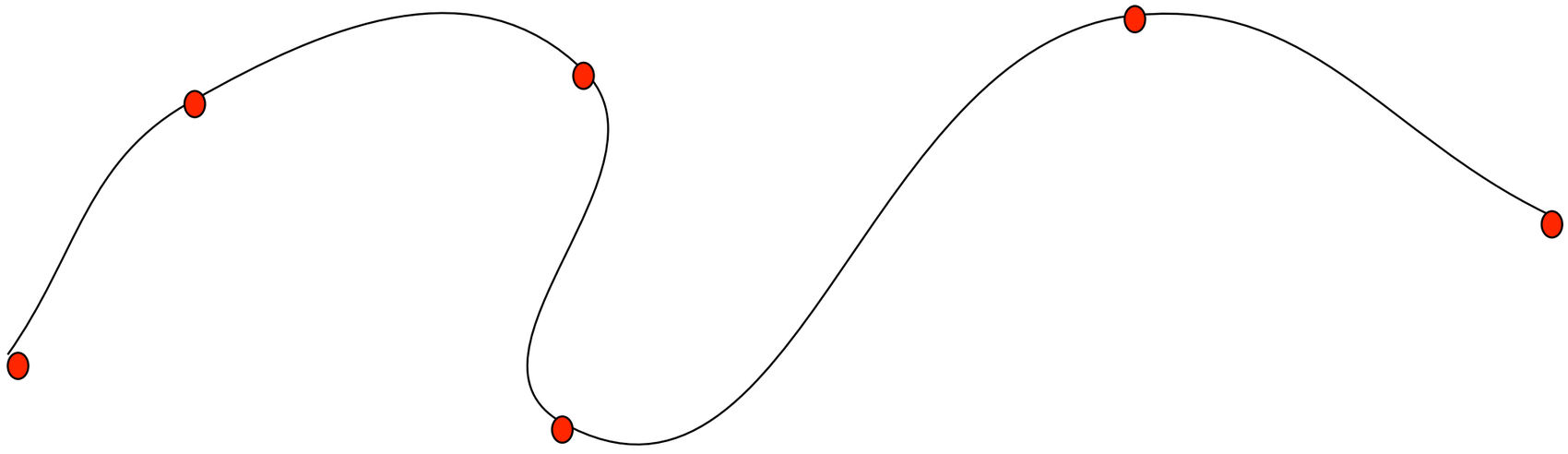
- Designing smooth curves (common in 2D illustrations)
- Filter design (e.g. cubic spline)
- Modelling smooth surfaces
- Representating of smooth surfaces (converted to polygons in real-time)
- Animation paths

Important application of splines: Text rendering!

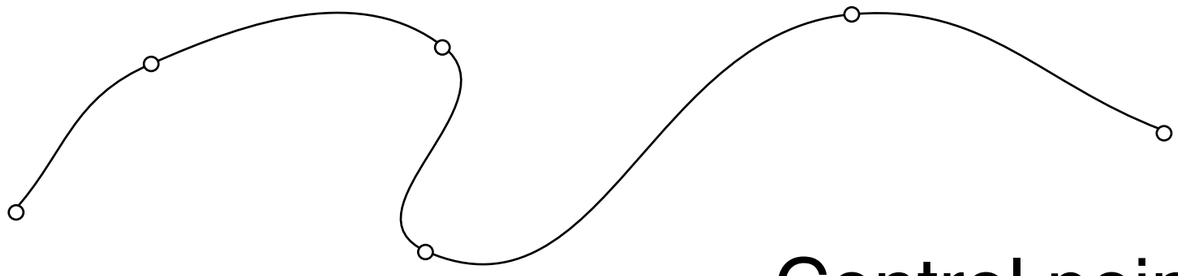


Control points

A spline is specified by a set of control points.

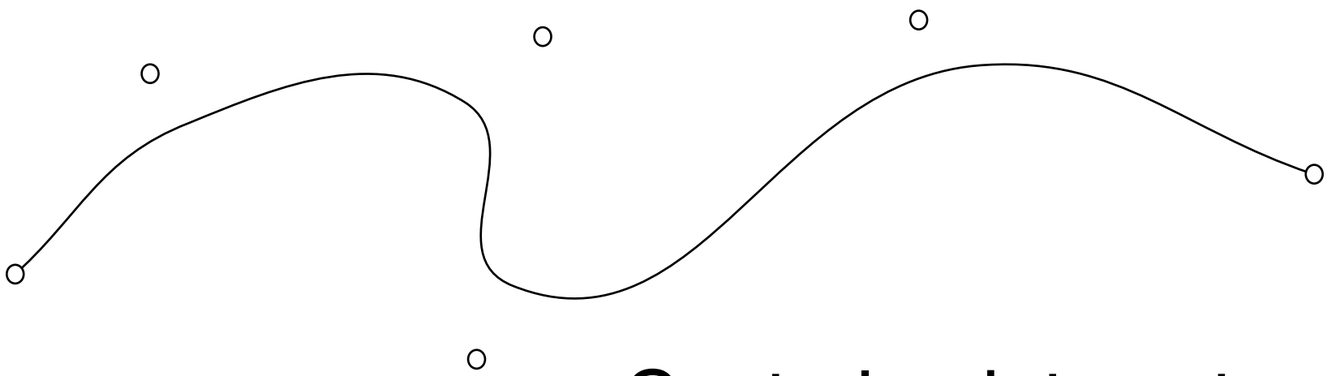


Interpolation spline



Control points on the curve.

Approximation spline



Control points not on the curve.

Parametric representation

$$x = x(u)$$

$$y = y(u) \qquad u_1 \leq u \leq u_2$$

$$z = z(u)$$

A set of functions for each coordinate

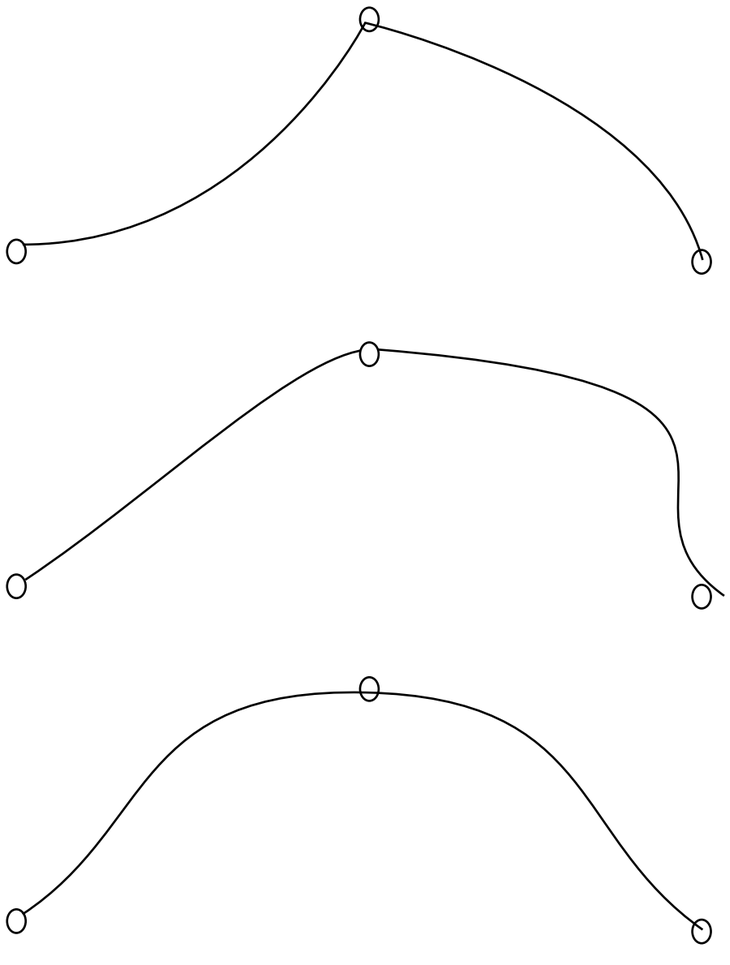
Spline continuity

Build curves from many splines

Joints between parts

Is the curve continuous, smooth?

Parametric continuity



C^0 = continuous position
= the curves meet

C^1 = continuous direction
= the curves meet at same angle

C^2 = continuous curvature
= the curves meet at same bend

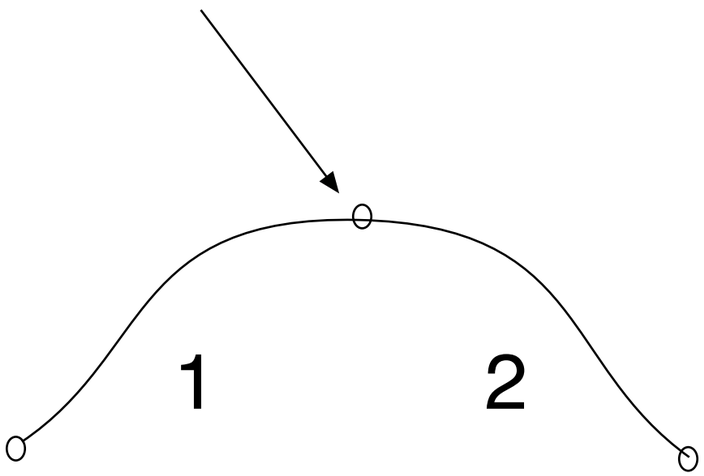
Specification of splines by polynomials in multiple sections

$$x_1(u) = a_{x1}u^3 + b_{x1}u^2 + c_{x1}u + d_{x1}$$

$$y_1(u) = a_{y1}u^3 + b_{y1}u^2 + c_{y1}u + d_{y1}$$

$$z_1(u) = a_{z1}u^3 + b_{z1}u^2 + c_{z1}u + d_{z1}$$

Continuity in this point?

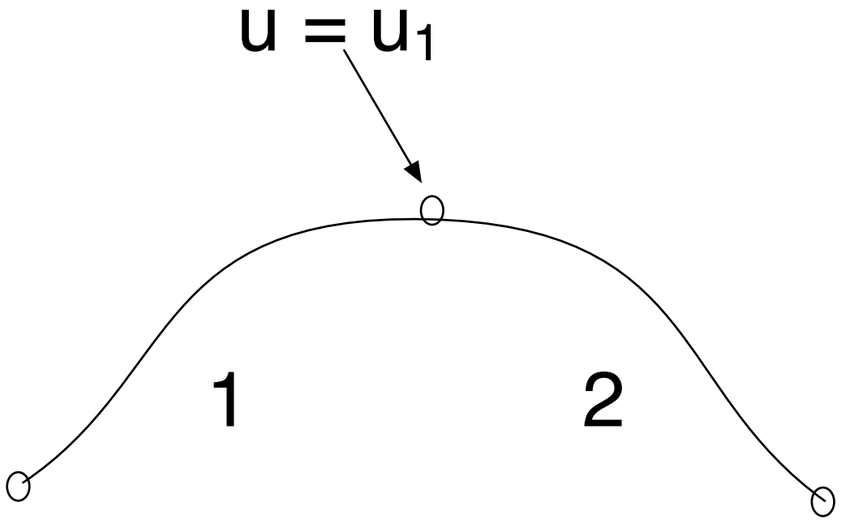


$$x_2(u) = a_{x2}u^3 + b_{x2}u^2 + c_{x2}u + d_{x2}$$

$$y_2(u) = a_{y2}u^3 + b_{y2}u^2 + c_{y2}u + d_{y2}$$

$$z_2(u) = a_{z2}u^3 + b_{z2}u^2 + c_{z2}u + d_{z2}$$

Parametric continuity as functions



C^0:

$$x_1(u_1) = x_2(u_1)$$
$$y_1(u_1) = y_2(u_1)$$
$$z_1(u_1) = z_2(u_1)$$

C^1:

$$x'_1(u_1) = x'_2(u_1)$$
$$y'_1(u_1) = y'_2(u_1)$$
$$z'_1(u_1) = z'_2(u_1)$$

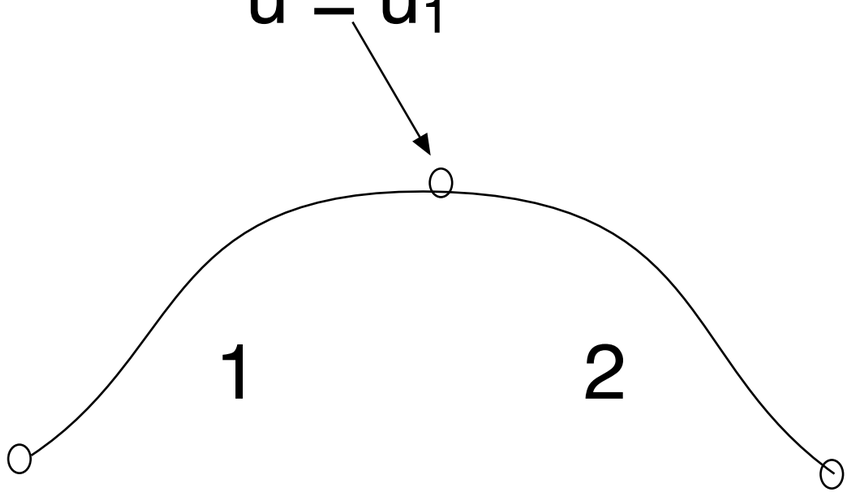
Geometric continuity

$G^0 = C^0$ = continuous position
= the curves meet

G^1 = proportinal direction
= the curves meet at same angle
but not same velocity

G^2 = proportional curvature
= the curves meet at same bend but
not same velocity

Geometric continuity as functions



G^0:

$x_1(u_1) = x_2(u_1)$
$y_1(u_1) = y_2(u_1)$
$z_1(u_1) = z_2(u_1)$

G^1:

$x'_1(u_1) = k*x'_2(u_1)$
$y'_1(u_1) = k*y'_2(u_1)$
$z'_1(u_1) = k*z'_2(u_1)$'
for some k

Essentially one less constraint

Blending functions

Rewrite parametric form to a set of polynomials, one polynomial for each control point

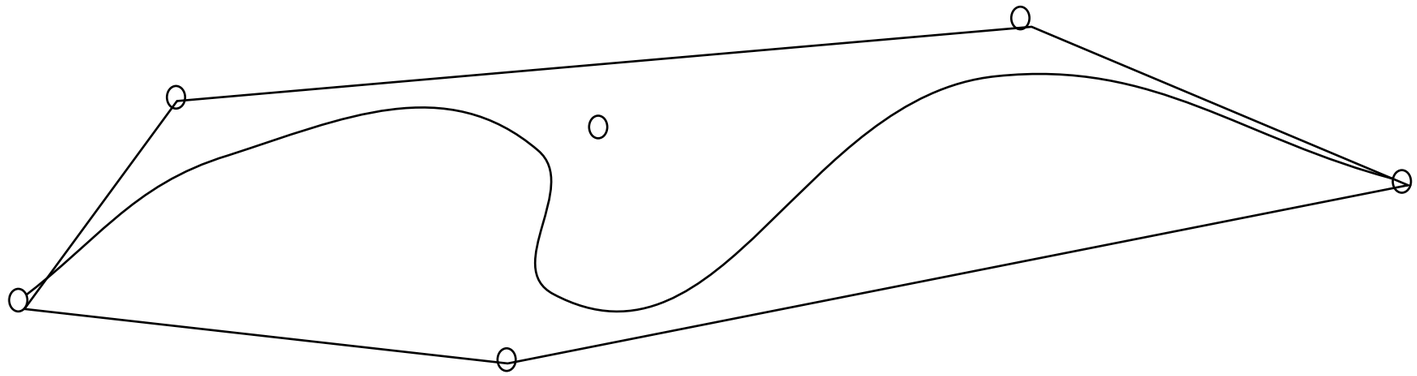
Approximation splines

Use a set of blending functions to blend together control points to points on the curve

Bézier curves
B-splines
NURBS

Common demand on approximations splines:

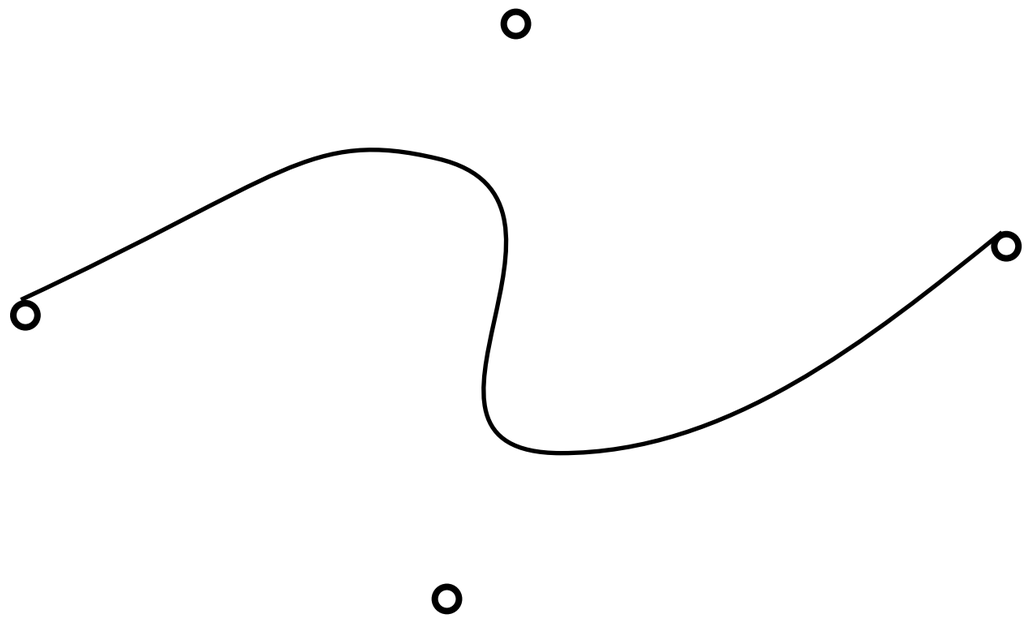
Stay within the convex hull of the control points!



Convex hull = minimal convex polygon enclosing a specified set of points

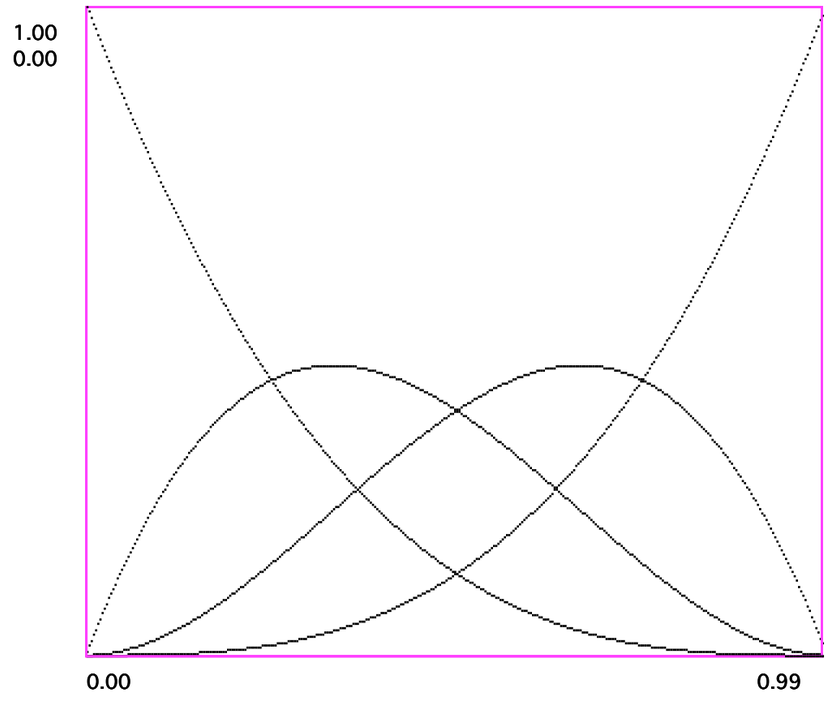
Bézier curve control points

Typically uses 3 or 4 control points per section



Blending Bézier curves

The 4 points are blended together using 4 blending functions



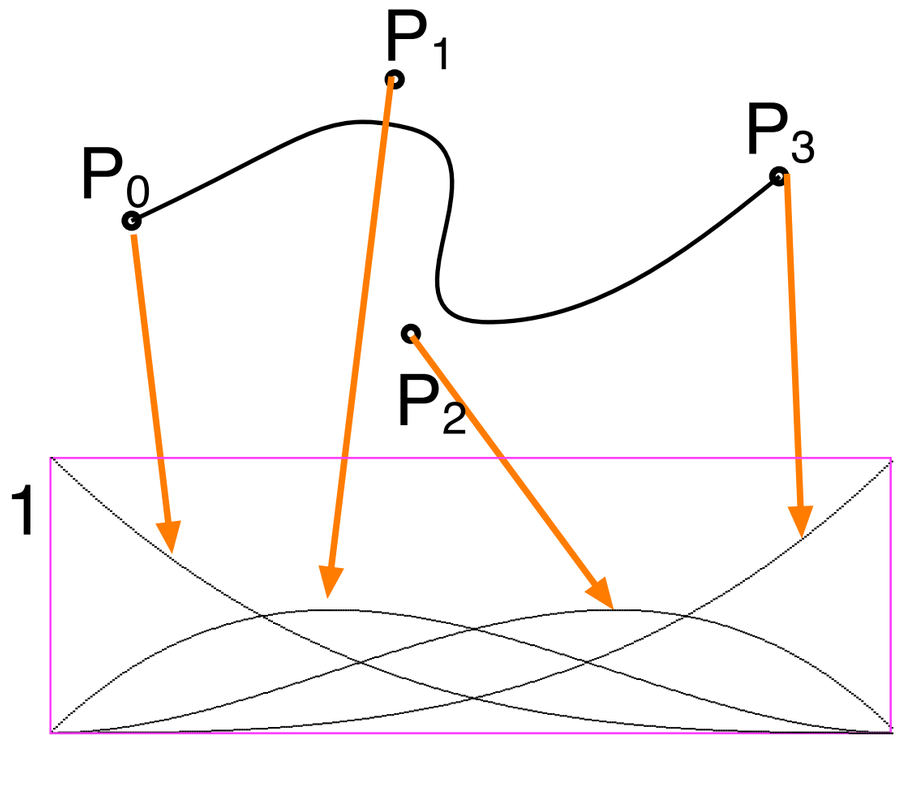
4 blending functions = Cubic Bézier

Bézier curves by Bernstein polynomials

Blending functions:
Bernstein polynomials

$$BEZ_{0,3} = (1-u)^3$$
$$BEZ_{1,3} = 3u(1-u)^2$$
$$BEZ_{2,3} = 3(1-u)u^2$$
$$BEZ_{3,3} = u^3$$

The sum is 1 for any u



$$BEZ_{0,3} = (1-u)^3$$
$$BEZ_{1,3} = 3u(1-u)^2u$$
$$BEZ_{2,3} = 3(1-u)u^2$$
$$BEZ_{3,3} = u^3$$

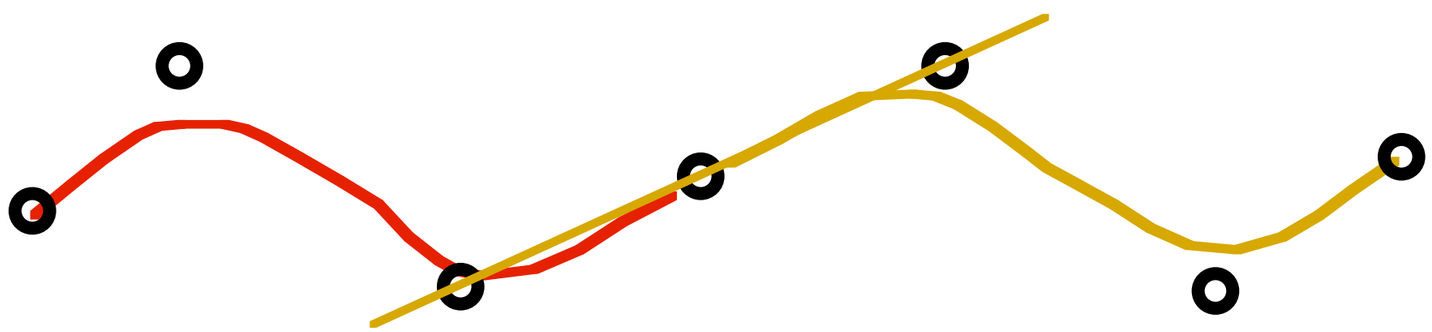
$$P(u) = P_0*(1-u)^3 + P_1*3u(1-u)^2 + P_2*3(1-u)u^2 + P_3*u^3$$

$$= \sum_{i=0}^{3} P_i * BEZ_{i,3}(u)$$

Fitting together sections

C^0/G^0 continuity: just fit the points

C^1 continuity: Tangents are equal along the edge.
G^1 continuity: Tangents have same direction along the edge.
Simple method: Put 3 points in a line



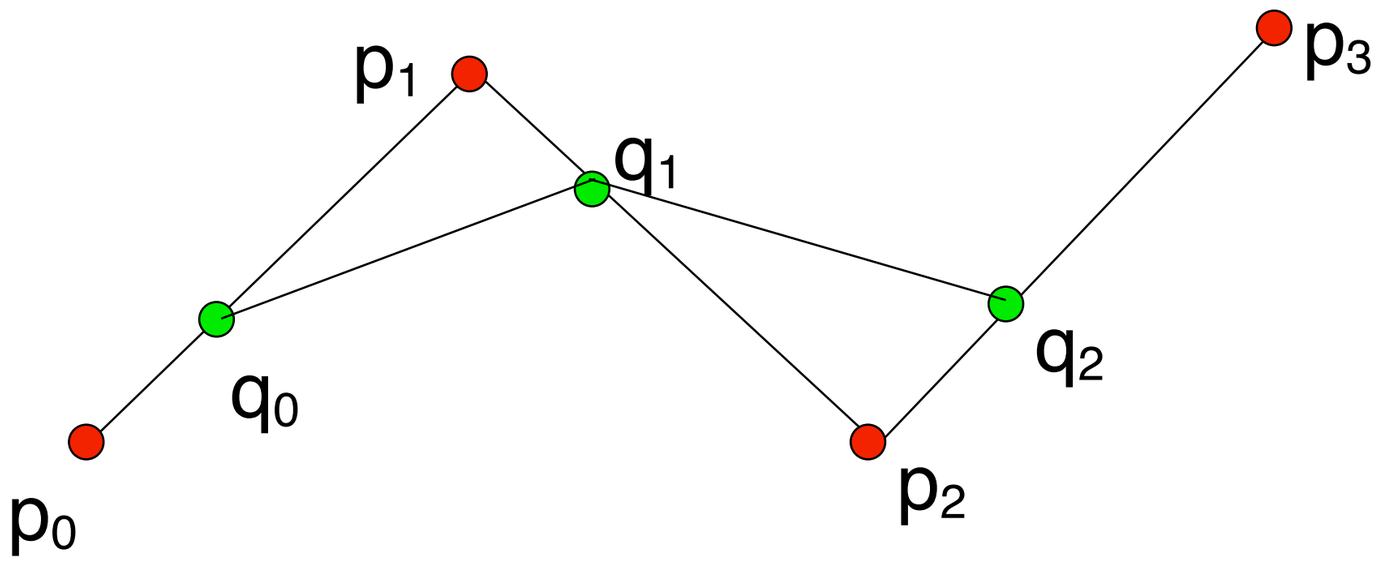
Quadratic Bezier curves

Three control points
2nd order polynomials

$$p(u) = (1-u)^2 p_0 + 2u(1-u)p_1 + u^2 p_2$$

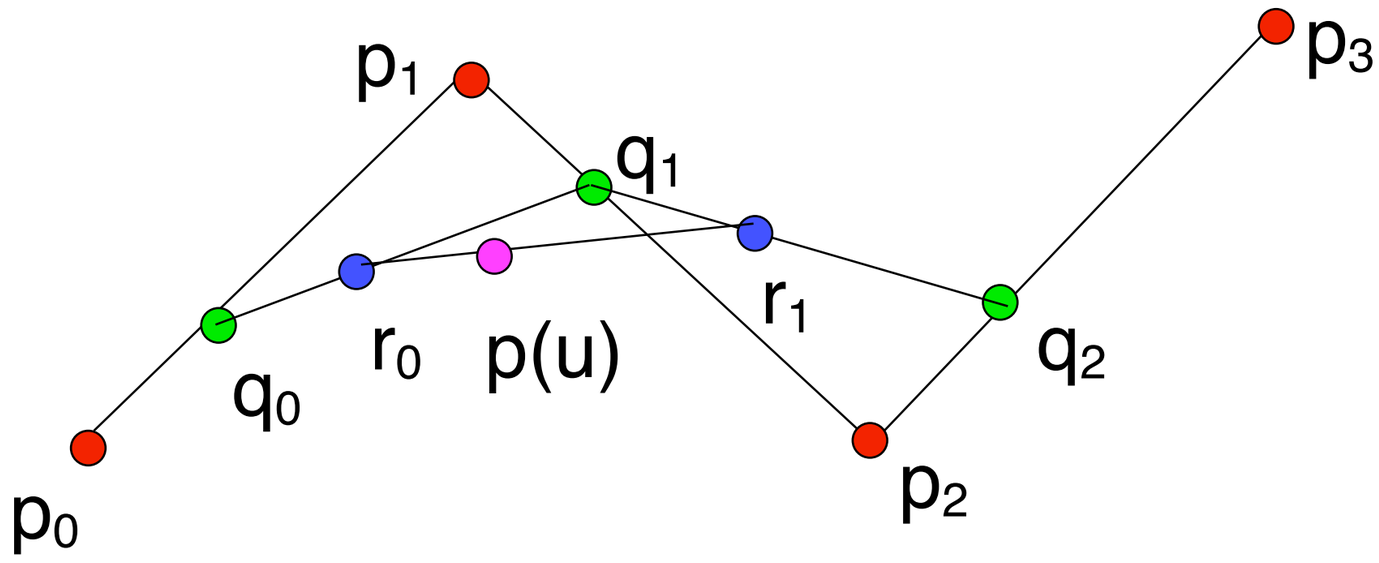
de Casteljau's algorithm

A Bézier is really an interpolation of interpolations!



de Casteljau's algorithm

Linear interpolations of linear interpolations until only one point remains



de Casteljau's algorithm

Gives us the Bernstein polynomials of any level we want.

Linear (2 points) = plain interpolation
Quadratic Béziers (3 points)
Cubic (4 points)
Higher levels possible but not practical

de Casteljau's algorithm, features

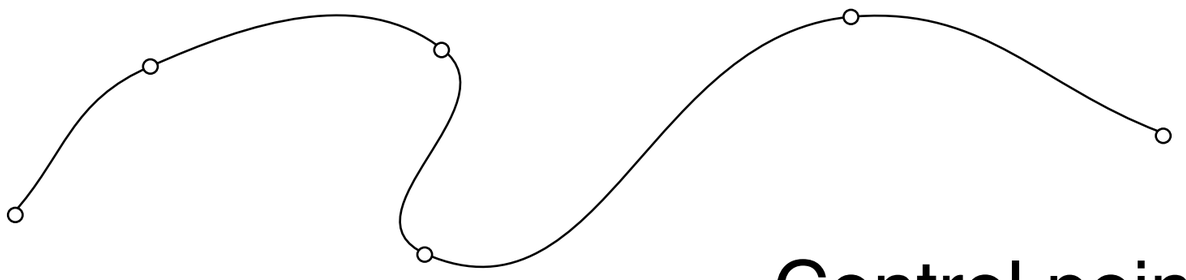
Obvious from figure/method:

- Bézier is always inside convex hull

- Fit together sections by keeping points along a line also obvious - we must start along the tangent!

Interpolation splines

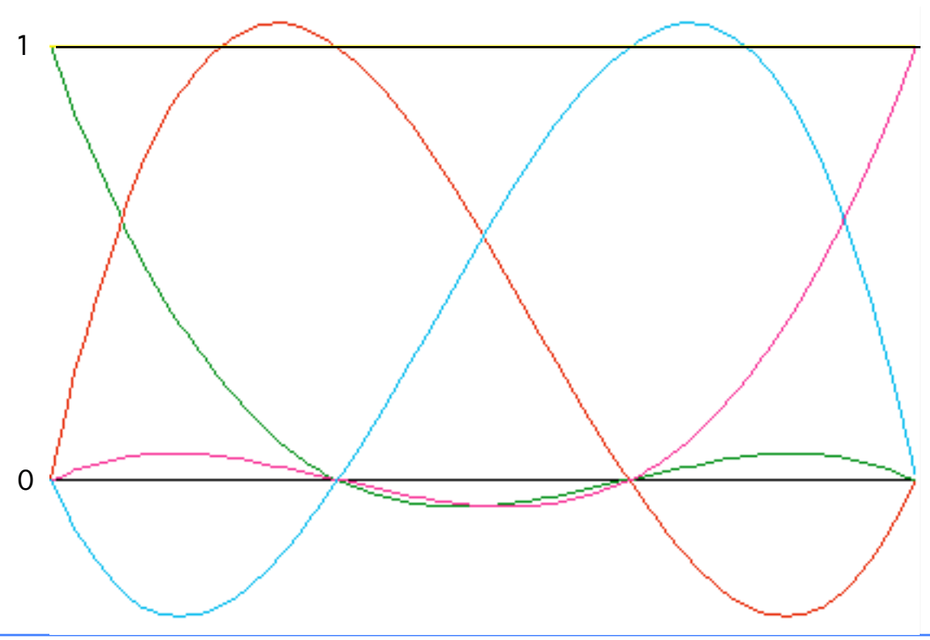
Passes through all control points.



Control points on the curve.

Blending functions for interpolation spline

All blending functions are zero or 1 at the control points!



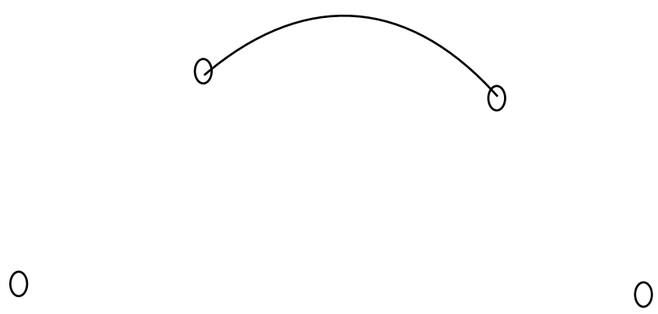
Actual blending functions for interpolated spline of 4 control points (similar to Bézier)

Catmull-Rom splines

Interpolation spline

Specified only by control points

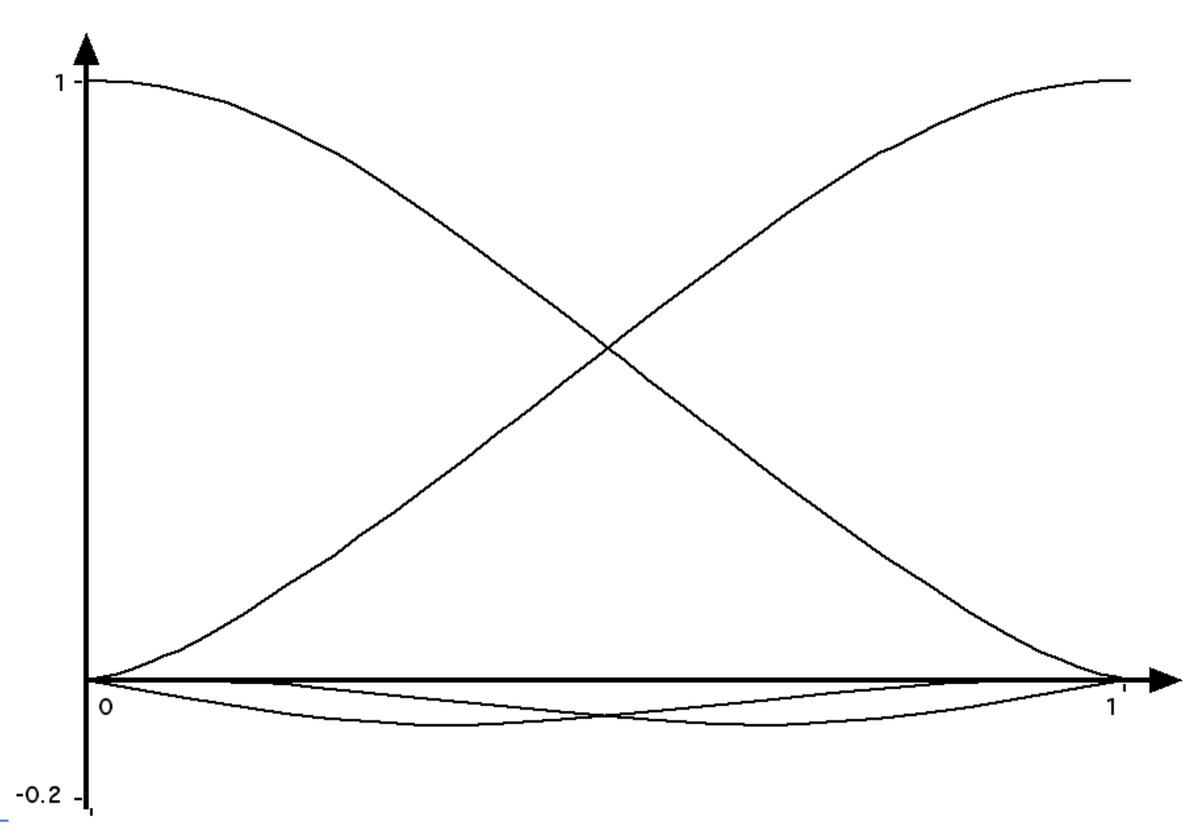
Calculated from 4 control points,
define between the middle two!



Catmull-Rom splines,
Blending functions

$$
\begin{aligned}
P(u) = \quad & p_{k-1}\,(-u^3/2 + u^2 - u/2) + \\
& p_k\,(3u^3/2 - 5u^2/2 + 1) + \\
& p_{k+1}\,(-3u^3/2 + 2u^2 + u/2) + \\
& p_{k+2}\,(u^3/2 - u^2/2)
\end{aligned}
$$

Catmull-Rom splines,
Blending functions



Application of Catmull-Rom splines:

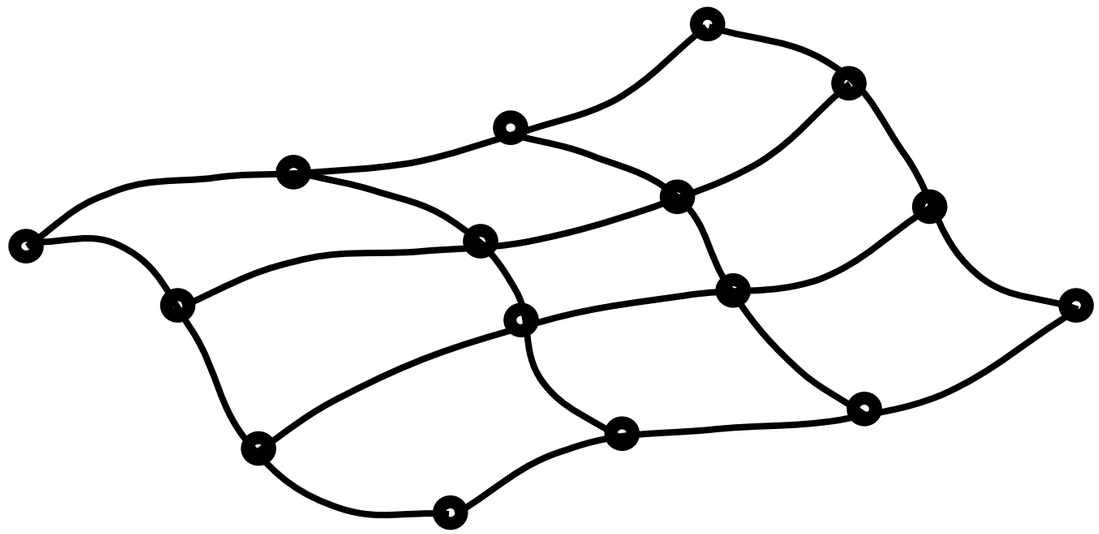
Animation paths!

- Defined by a sequence of points on the curve
  - Always G^1/C^1 continuous

Bézier surfaces

A surface is built from a set of Bézier patches

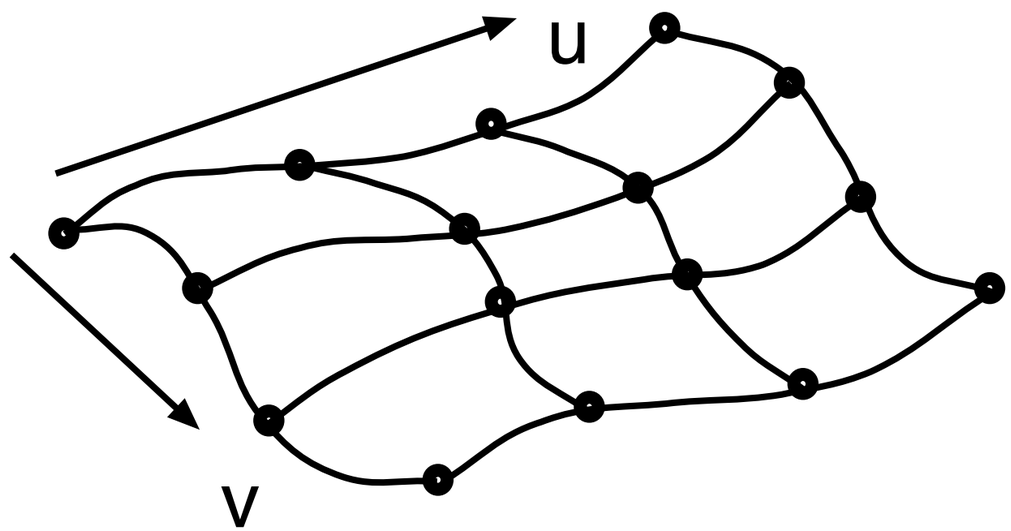
A Bézier patch consists of 16 control points in a 4x4 grid



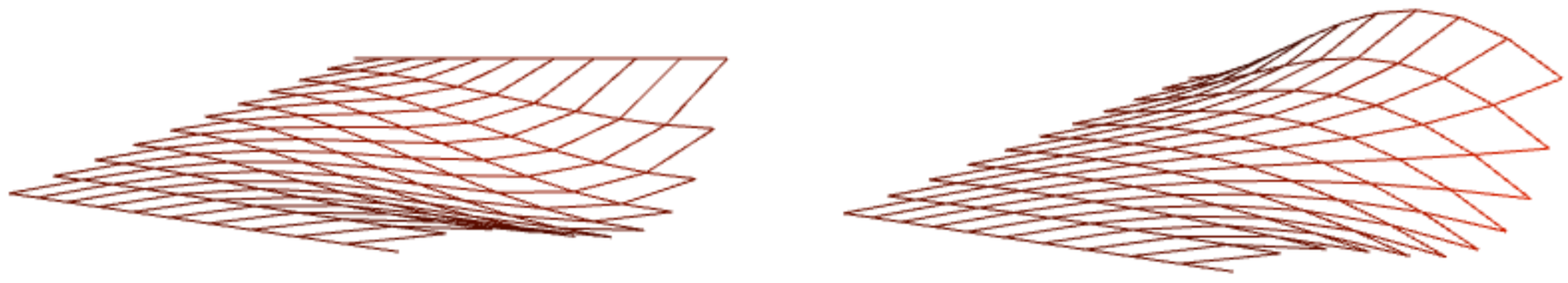
Bézier surfaces

Blending of the 16 control points as a 2-dimensional sum

$$P(u,v) = \sum_{j=0}^{3} \sum_{k=0}^{3} p_{j,k}\, BEZ_{j,3}(v)\, BEZ_{k,3}(u)$$



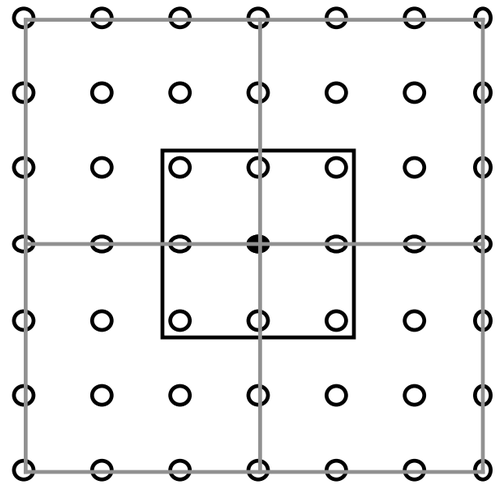
Bézier surface example



Fitting together patches

Fit in both u and v direction

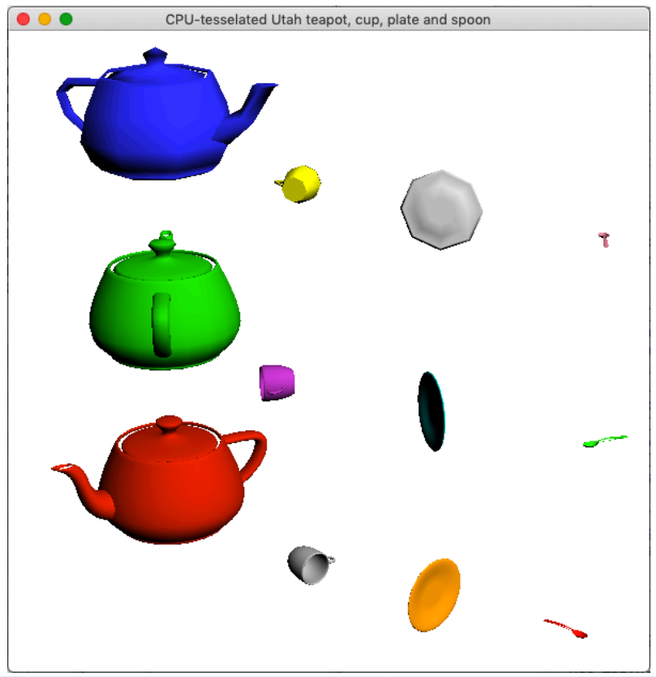
Make a 3x3 “joystick” at each corner

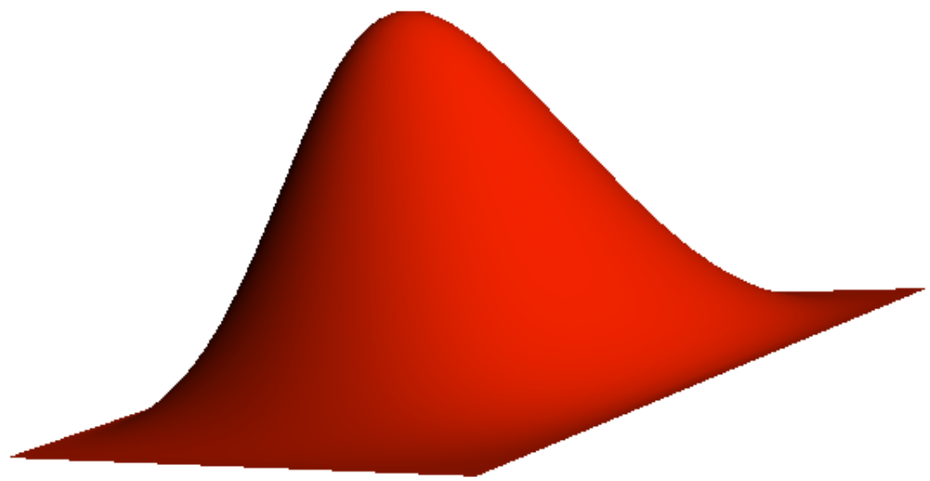


Bézier surface examples using GLUGG

GLUGG supports Bézier surfaces:

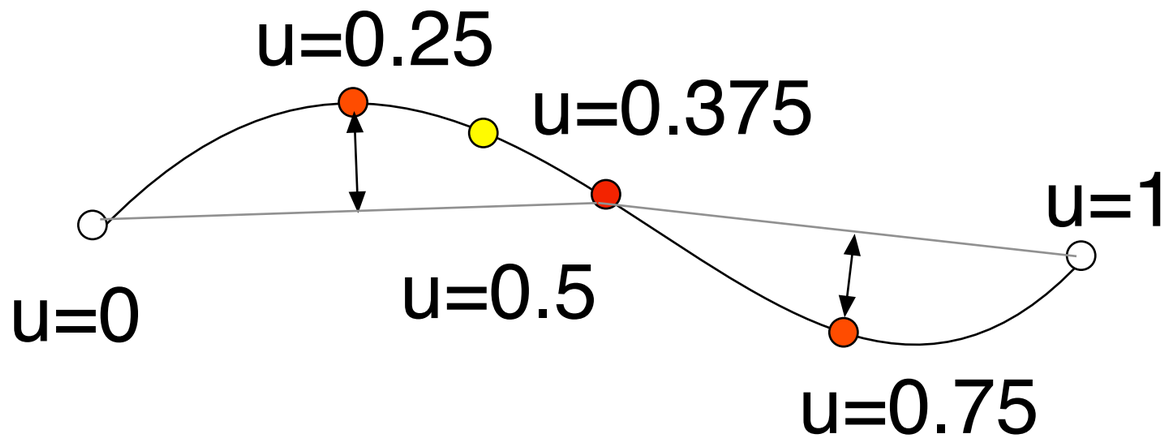
```
vao[0] = gluggBuildBezierPatchModel(teapotVertices,
teapotIndices, 0, 32, &triangles[0], program, kStep);
```





Drawing splines

Subdivide the spline until the error is small enough.



Or a number of uniform steps depending on size!

More object representation soon...

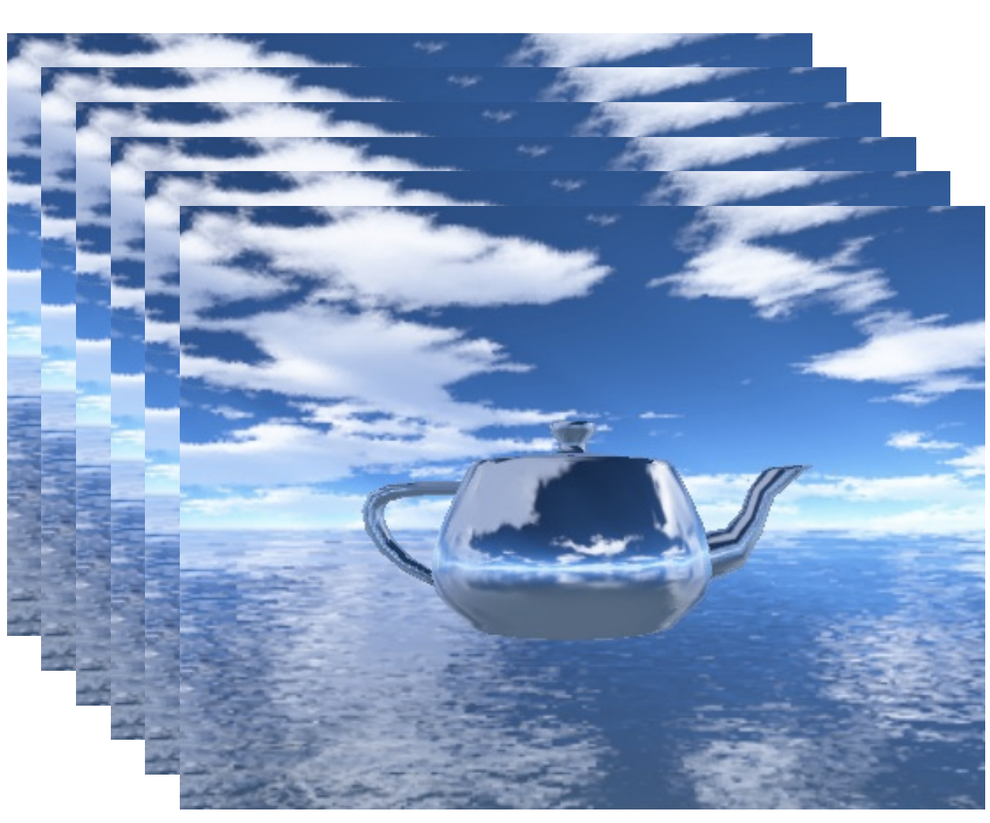
Fractals and procedural generation

but first

let us move to the animation subject!

Animation

Essentially a question of flipping between many still images, fast enough



Animation as a topic

- Page flipping, double-buffering
- Sprite animation
- Movement and posing
- Collision detection and handling
- Deformations

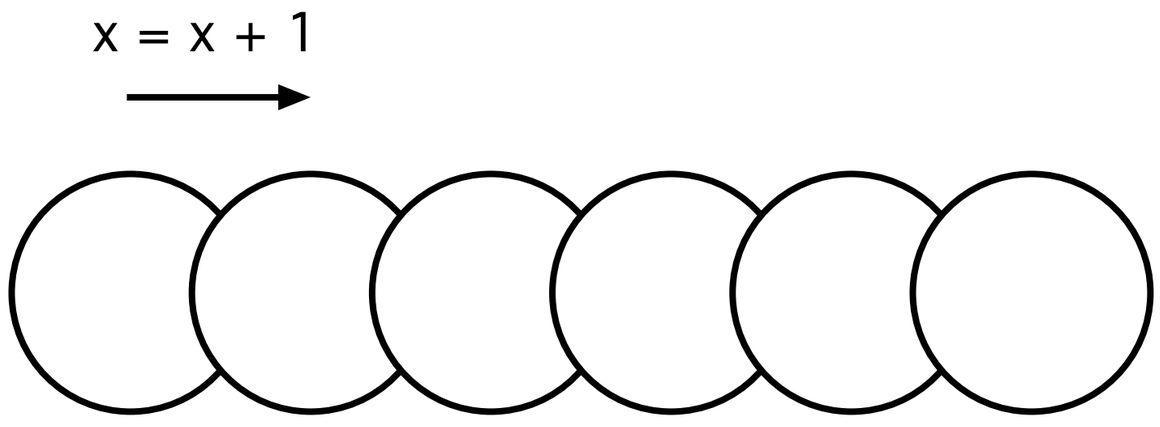
Animation techniques for moving objects

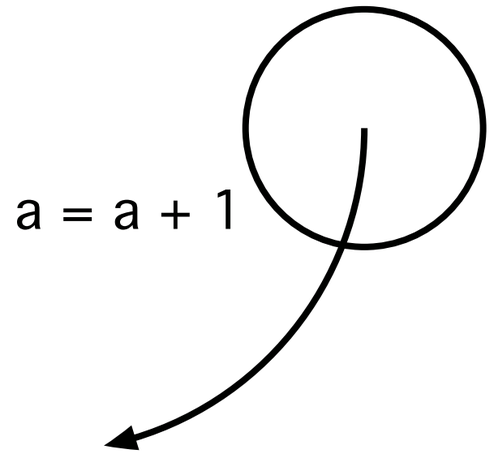
- Procedural animation
- Physics-based animation
- Pre-programmed animation paths

Procedural animation

Simple frame by frame movement following some procedural rule.

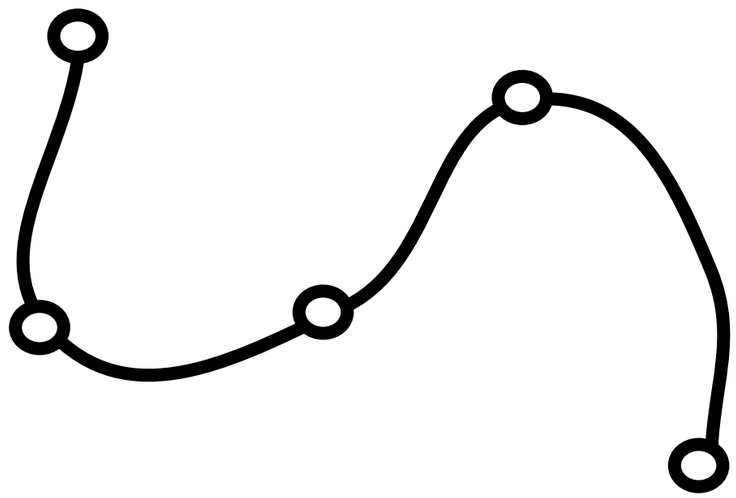
What we are already doing in the labs!

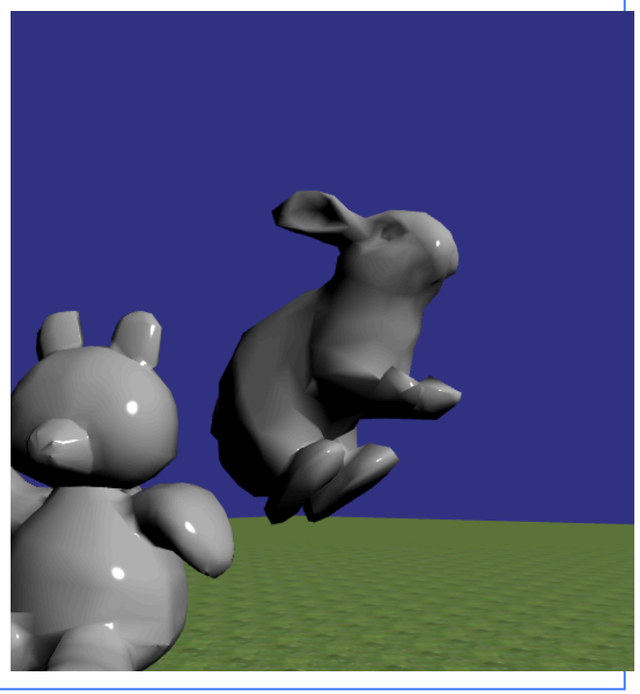




Animation paths

Use Catmull-Rom splines! Predictable, smooth, continuous!





Character animation

- Pre-defined poses
- Key-frame animation
- Forward kinematics
- Inverse kinematics
- Physics based animation
- Motion capture

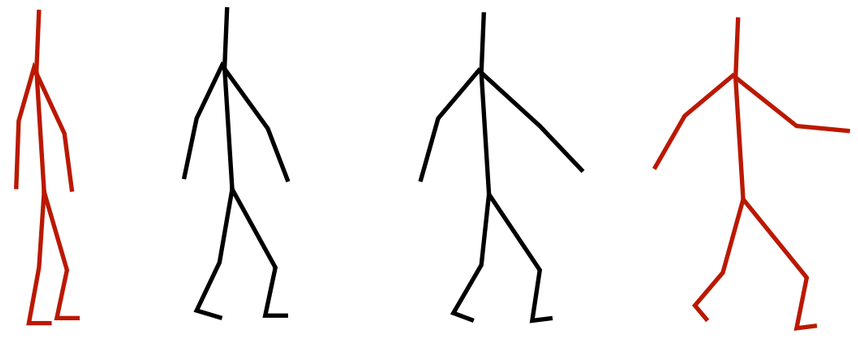
Key-frame animation

Fewer pre-defined poses

Key-frames are designed at suitable intervals

Frames between keyframes are interpolated (morphed)

Very common method for real-time animation



Kinematics

How to find these poses!

Kinematics = movement without forces

Forward kinematics:

Specify poses by specifying rotation of joints. Easy to implement, but specifying poses is much trial-and-error.

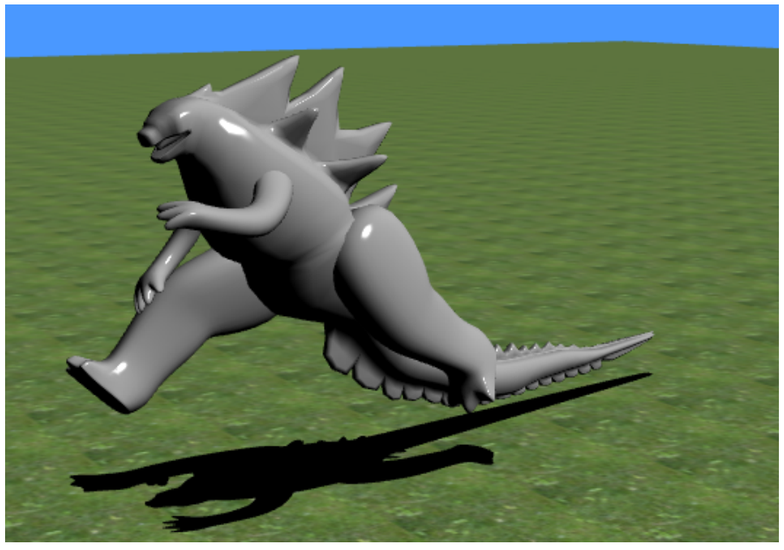
Inverse kinematics:

Goal-driven posing. Specify where some part should go (i.e. a hand) and calculate necessary rotations

Character animation in this course
(using the tools central to the course)

Animation of rigid parts (robot style/windmill style)

Model in lab material: Godzilla model!



Particle systems

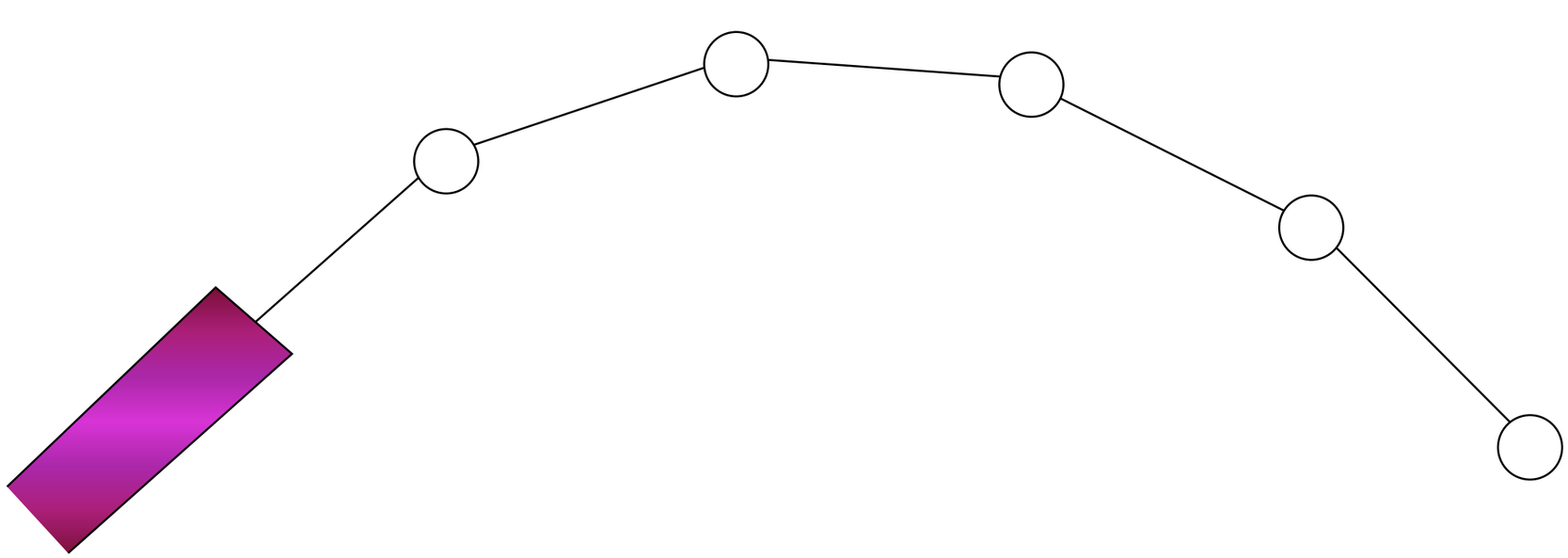
Spectacular effects with little effort!

Many small moving objects.

- Explosions
- Water
- Fire
- Snow
- Rain

Particle system

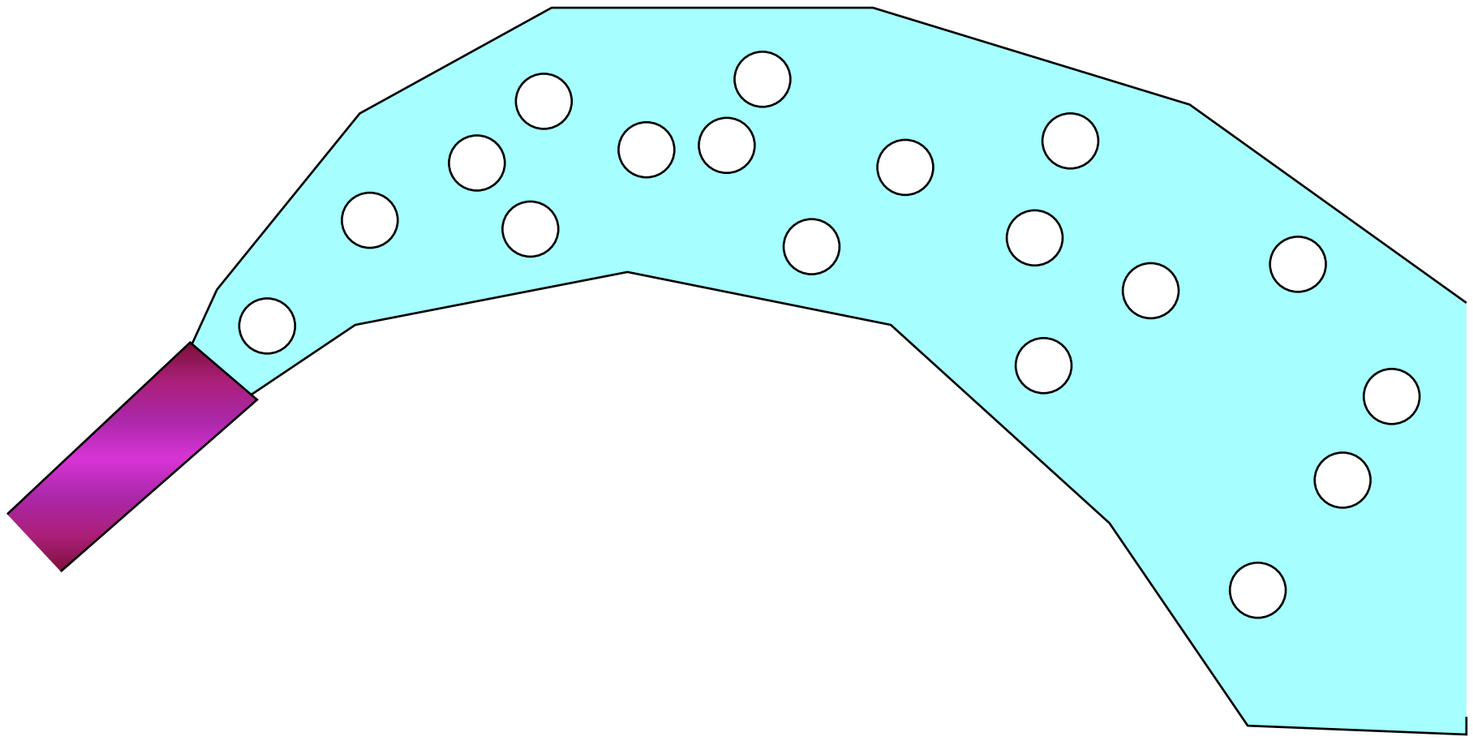
Example: Water



No randomness - bad

Particle system

Example: Water



Life of a particle

- Initial position
- Initial speed (usually with some randomness))
- Movement (usually independent, physically realistic)
- Termination rule (e.g. hits ground, fades away after some time...)

Particle physics

Point-based physics usually fine

Movement according to fundamental physics:

acceleration = gravity + forces/mass
speed = speed + acceleration
position = position + speed

“Euler integration”

Particle system on CPU

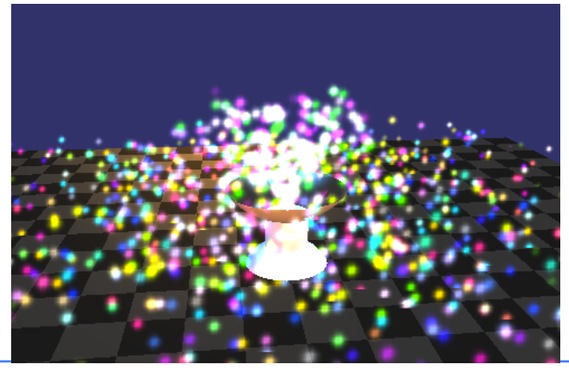
CPU-driven particle systems OK up to a certain size

Make arrays of positions and velocity.

Data transfer (new positions) of all particles can be a bottleneck. Multiple calls for each?

How much can we compute on the GPU?

This was not a problem:



Drawing particle systems

Large number of very simple models (billboards)!

Modest demands on GPU, but very large number of function calls!

Solution: Instancing

Instancing, revisited

Draw a large number of the same model!

Each instance has an index, the instance number.

glDrawArraysInstanced(GL_TRIANGLES, 0, 3, 10);

draws a triangle 10 times!

gl_InstanceID tells the shader which instance we have.
Use for affecting position.

Feeding data to instancing

Much data must be transferred with few function calls!

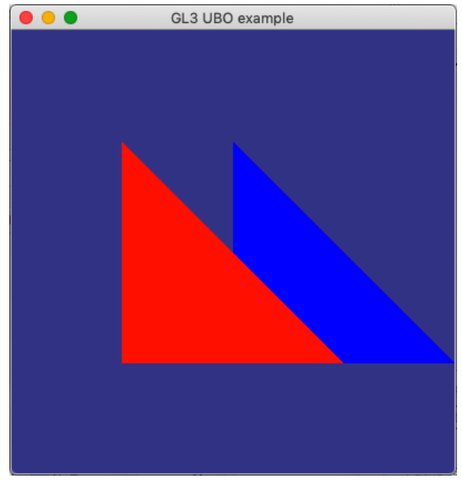
Possible methods:

Textures
UBO (Uniform Buffer Object)
Instanced arrays

UBO demo

One call for drawing 2 triangles as separate instances.

A buffer with two colors is indexed with the instance id.

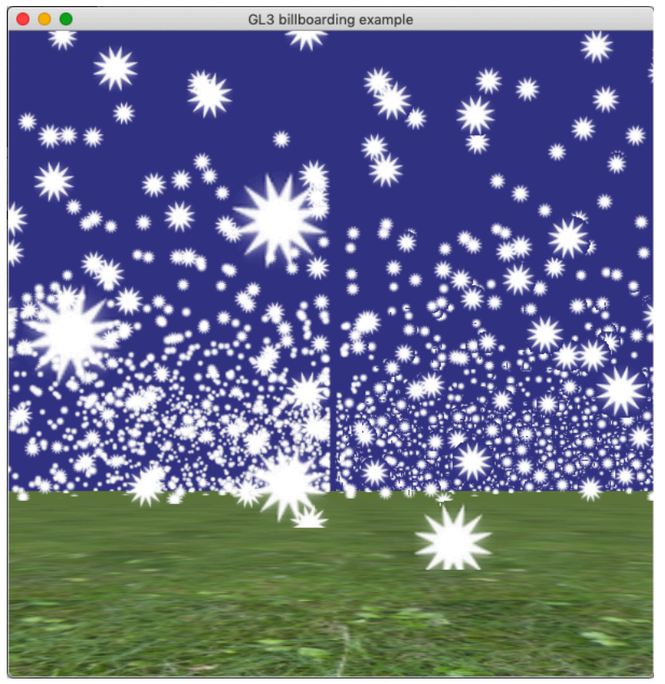


Demo: Data in texture

A texture containg random numbers is uploaded.

Positions are calculated from time and one texel per instance.

Make sure to hit the *center* of each texel!



Instanced arrays

An array passed one item at a time, like vertex attributes

Uses glVertexAttribDivisor to split the array in instances

Conclusion: Several ways to get that info in!

Advanced particle system techniques

Particle systems on GPU

Texture based particle systems

Separate compute kernels

Randomness on GPU

TSBK03

Particle system on GPU

CPU-driven particle systems OK up to a certain size

Data transfer (new positions) of all particles can be a bottleneck

Can the whole particle system be computed on the GPU?

TSBK03

Texture based particle systems

Use textures to store x, y, z, dx, dy, dz

Store as color components (r, g, b)

Needs advanced texturing features (render to texture, floating-point buffers)

Particles as billboards. Each polygon must identify its particle data.

TSBK03/
TDDD56

Separate compute kernels for particle systems

CUDA, OpenCL, Compute shaders

Free choice of data formats

Less integration with the OpenGL pipeline

TSBK03/
TMN084

Randomness in GPU

The GPU has no random number generator!

Rain example: Random numbers in texture generated on CPU.

Mathematical calculation in shader is possible. (Other course but ask me as needed.)

Next time:

Collision detection and handling