Computer Graphics

Curves (conclusion).
Some procedural techniques.

Konstantin Tretyakov
kt@ut.ee

Dec 11, 2013

Previous time

- Polynomial curve:

$$\mathbf{p}(t) = \mathbf{c}_0 + \mathbf{c}_1 t + \cdots + \mathbf{c}_n t^n := \mathbf{C}\mathbf{T}_n(t)$$

- Representation via geometry and basis matrices

$$\mathbf{p}(t) = \mathbf{G}\mathbf{M}\mathbf{T}(t)$$

- Representation via blending functions

$$\mathbf{p}(t) = \sum_{i=0}^{n} b_i(t)\mathbf{p}_i, \qquad \sum_{i=0}^{n} b_i(t) = 1$$

Previous time: Curves

- Interpolating
  - Lagrange (not much used)
  - Natural spline (CAD/CAM, trajectories)
- Approximating
  - Bezier' (Photoshop/GIMP/MSWord, …)
  - B-spline (trajectories)
  - NURBS (CAD/CAM, Blender/Maya, …)

Quiz

- Devise a *Rational Bezier curve*.

Quiz

- Devise a *Rational Bezier curve*.

Bezier curve:

$$\boldsymbol{p}(t) = \sum_i B_i^n(t)\boldsymbol{p_i}$$

where

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

Quiz

- Devise a *Rational Bezier curve*.

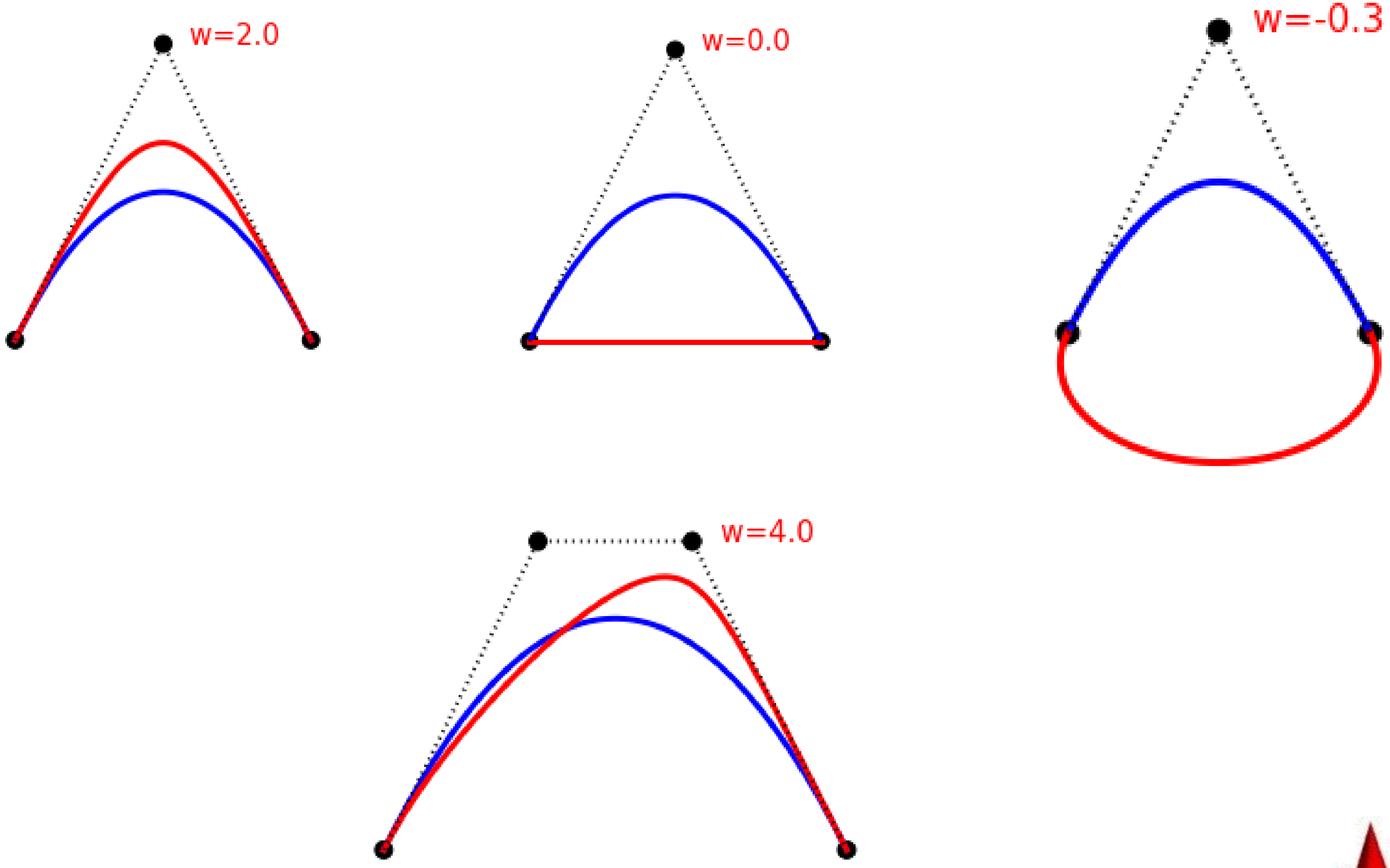
Rational Bezier curve:

$$\boldsymbol{p}(t) = \frac{\sum_i B_i^n(t) w_i \boldsymbol{p_i}}{\sum_i B_i^n(t) w_i}$$

where

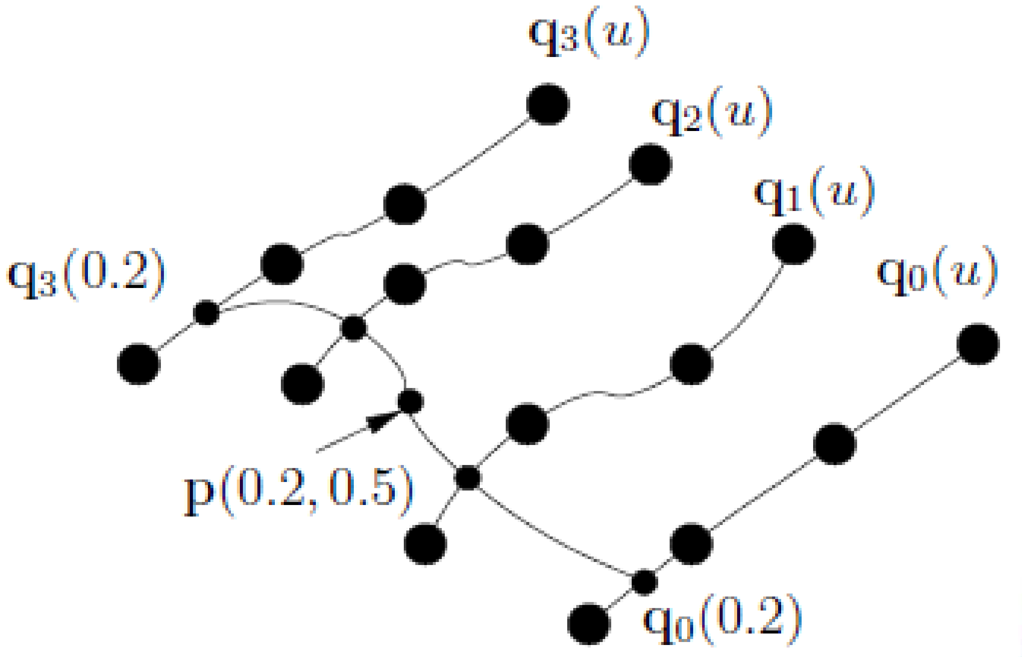
$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

Rational Bezier curves



Previous time: Surfaces

Tensor product surfaces



$$\boldsymbol{p}(u,v) = \sum_i \sum_j b_i(u) b_j(v) \boldsymbol{p}_{ij}$$

Quiz

- Consider a 1-degree polynomial parametric tensor product surface.
- How many control points do you need?
- What are the blending functions?

Quiz

- Consider a 1-degree polynomial parametric tensor product surface.
- How many control points do you need?
  - 2 x 2 = 4
- What are the blending functions?
  - $b_{00}(u, v) = (1 - u)(1 - v)$
  - $b_{01}(u, v) = (1 - u)v$
  - $b_{10}(u, v) = u(1 - v)$
  - $b_{11}(u, v) = uv$

Next

- **B-spline. Non-uniform B-spline.**
- **Rational B-spline. NURBS.**
- **Surfaces. Tensor product surfaces.**
- **Rendering curves and surfaces.**
- **Curves, surfaces & OpenGL.**

Rendering curves & surfaces

- Three main techniques:
  - Conversion to a fixed-size mesh
  - Recursive subdivision
  - Raytracing

Conversion to a fixed-size mesh

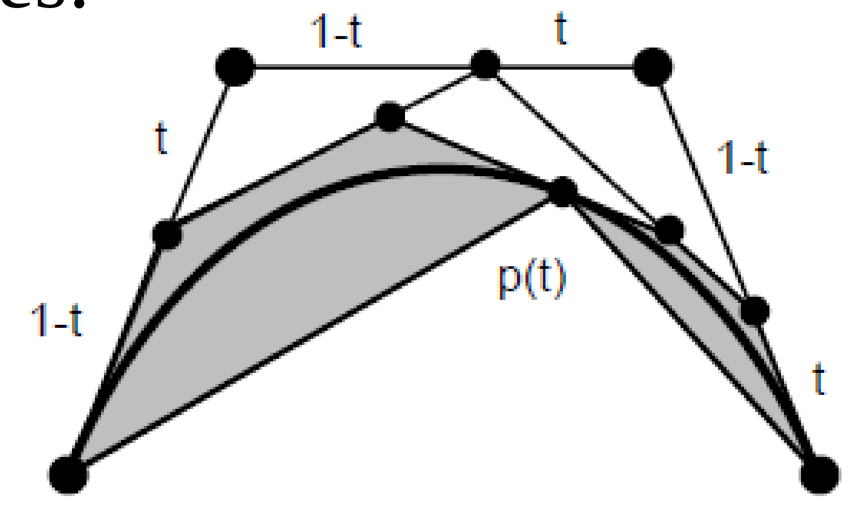
- Given a curve $\boldsymbol{p}(t)$ or a surface $\boldsymbol{p}(u, v)$ we can simply compute its points for a regular grid, creating a mesh.
- E.g. for a surface, use vertices

$$\boldsymbol{v}_{ij} = \boldsymbol{p}\left(\frac{i}{n}, \frac{j}{n}\right)$$

- Main drawback: we need to prespecify the resolution of the mesh. The resolution is uniform over all surface/curve.

Recursive subdivision

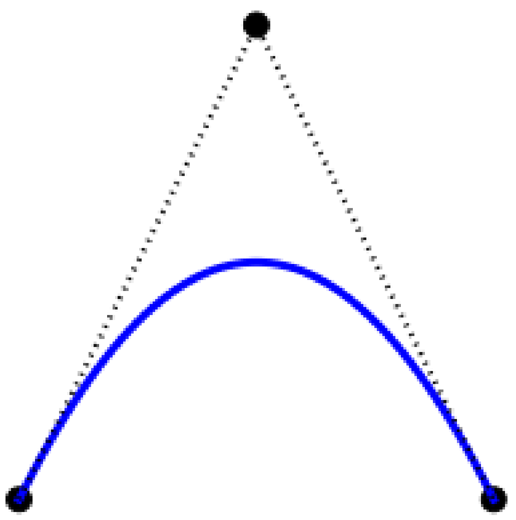
A Bezier curve can be efficiently split into two pieces:



We can perform such subdivision recursively until each piece is small enough or straight enough to be rendered as a line segment.

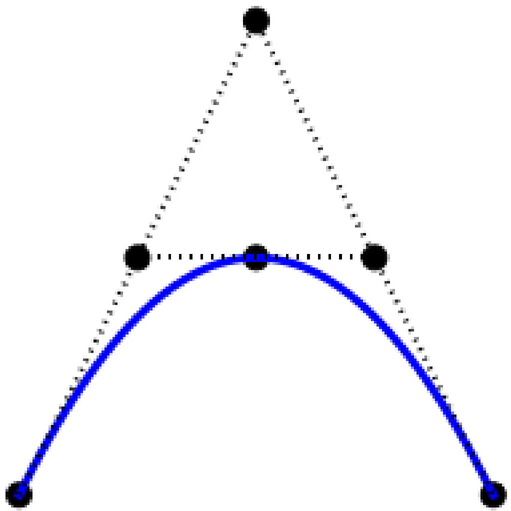
Quiz

Given a quadratic Bezier curve on the following three control points, split it into two smaller quadratic curves at point $\boldsymbol{p}(0.5)$.



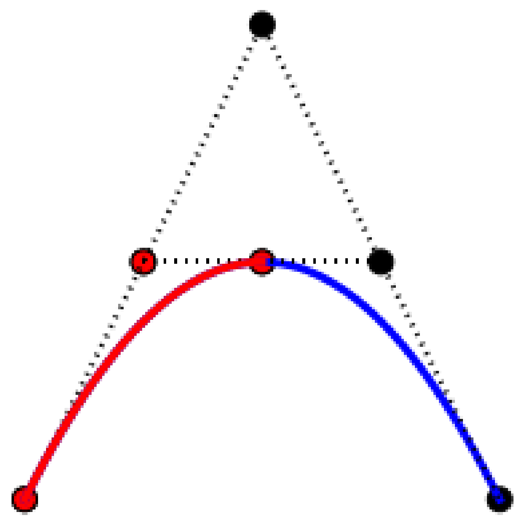
Quiz

Given a quadratic Bezier curve on the following three control points, split it into two smaller quadratic curves at point $\boldsymbol{p}(0.5)$.



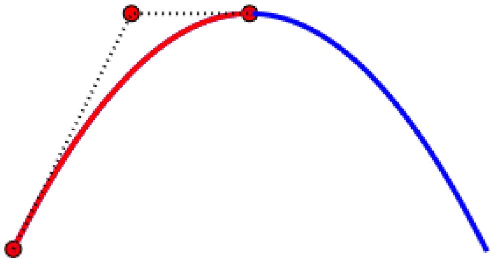
Quiz

Given a quadratic Bezier curve on the following three control points, split it into two smaller quadratic curves at point $\boldsymbol{p}(0.5)$.



Quiz

Given a quadratic Bezier curve on the following three control points, split it into two smaller quadratic curves at point $\boldsymbol{p}(0.5)$.



Quiz

Recursive subdivision only works for Bezier' curves. What do we do for other curves?

Quiz

Recursive subdivision only works for Bezier' curves. What do we do for other curves?

Any other curve can be converted to a Bezier' curve!

Converting curves

Let the control points of a cubic B-spline patch be given in the geometry matrix P_{BS}.

How can we find the corresponding Bezier' control points P_B?

Converting curves

Let the control points of a cubic B-spline patch be given in the geometry matrix P_{BS}.

How can we find the corresponding Bezier' control points P_B?

$$\mathbf{P}_{BS}\mathbf{M}_{BS}\mathbf{T}(t) = \mathbf{P}_B\mathbf{M}_B\mathbf{T}(t)$$

$$\mathbf{P}_{BS}\mathbf{M}_{BS} = \mathbf{P}_B\mathbf{M}_B$$

$$\mathbf{P}_B = \mathbf{P}_{BS}\mathbf{M}_{BS}\mathbf{M}_B^{-1}$$

Raytracing

- Finally, we can raytrace polynomial surfaces.
- This typically involves solving a quadratic or cubic equation.
- Consequently, *quadratic* patches are usually easier to raytrace.

OpenGL

- OpenGL 2.1-3.0 (or 3.1+ compatibility mode) supports rendering of Bezier' curves.
- Any other curves must be converted to Bezier form.
- NURBS curves are supported by the GLU library (which deals with the conversion to Bezier form).

OpenGL example

```
GLfloat control_points[4][3] =
  {{0, 0, 0},
   {1, 1, 0},
   {3, 1, 0},
   {4, 0, 1}};

glEnable(GL_MAP1_VERTEX_3);
glMap1f(GL_MAP1_VERTEX_3,
                0.0, 1.0,
                  3,   4,
                &control_points[0][0]);
```

Configuring the curve evaluator.

OpenGL example

Rendering the curve

```
glBegin(GL_LINE_STRIP);
  for (int i = 0; i <= 20; i++)
     glEvalCoord1f((Glfloat) i/20.0);
glEnd();
```

Equivalently:

```
glMapGrid1f(20, 0.0, 1.0);
glEvalMesh1(GL_LINE, 0, 20);
```

OpenGL example

A Bezier' evaluator may be used to not only generate vertices, but also normals, colors, texture coordinates, etc:

```
glEnable(GL_MAP1_COLOR_3);
glMap1f(GL_MAP1_COLOR_4,
                0.0, 1.0,
                  4,   4,
                &control_colors[0][0]);
```

Surfaces in OpenGL

Bezier' surfaces can be rendered in the same way. First initialize the "2D evaluator":

```
Glfloat ctrl_points[4][4][3] = ...;
glEnable(GL_MAP2_VERTEX_3);
glMap2f(GL_MAP2_VERTEX_3, 0, 1,   3, 4,
                          0, 1, 12, 4,
                &ctrl_points[0][0][0]);
```

Surfaces in OpenGL

… and then sample from it, either manually using `glEvalCoord2f`, or "automatically":

```
glMapGrid2f(10, 0.0, 1.0, 10, 0.0, 1.0);
glEvalMesh2(GL_FILL, 0, 10, 0, 10);
```

NURBS

```
GLUnurbsObj* nurb = gluNewNurbsRenderer();
gluNurbsProperty(nurb, GLU_DISPLAY_MODE, GLU_FILL);

gluBeginSurface(nurb);
gluNurbsSurface(nurb, 8, knots, 8, knots,
                12, 3, &control_points[0][0][0],
                 4, 4, GL_MAP2_VERTEX_3);
gluEndSurface(nurb);

gluDeleteNurbsRenderer(nurb);
```

Summary

- **Polynomial parametric curves and surfaces:**
  - Interpolating: Lagrange, Natural spline
  - Approximating: Bezier', B-spline, NURBS

- **Representation**: blending functions, basis matrix.

- **Rendering**: mesh conversion, subdivision, raytracing

- **OpenGL**: `glMap`, `glEvalCoord`, `glEvalMesh`, ...
- **GLU**: `gluNurbsSurface`, ...

Some procedural techniques

- Fractals
  - Fractal terrains
  - L-systems (aka Generative grammars)
- Particles
  - Fire, water and magic sparks
  - Boids

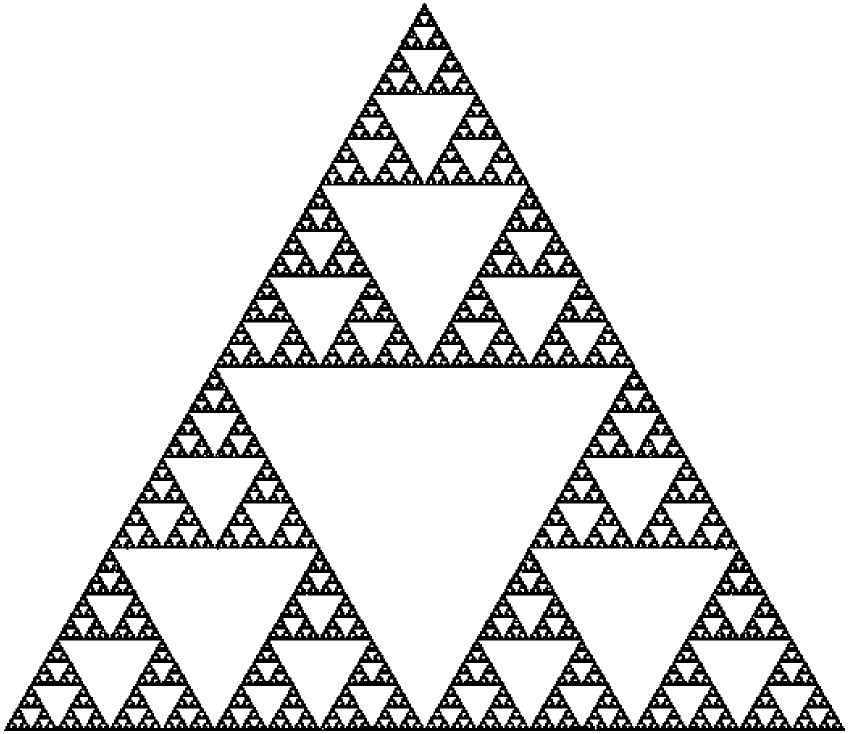
Fractals

- Fractals: shapes with self-similarity
  - Exact self-similarity
  - Stochastic self-similarity

http://en.wikipedia.org/wiki/Self-similarity

Fractals

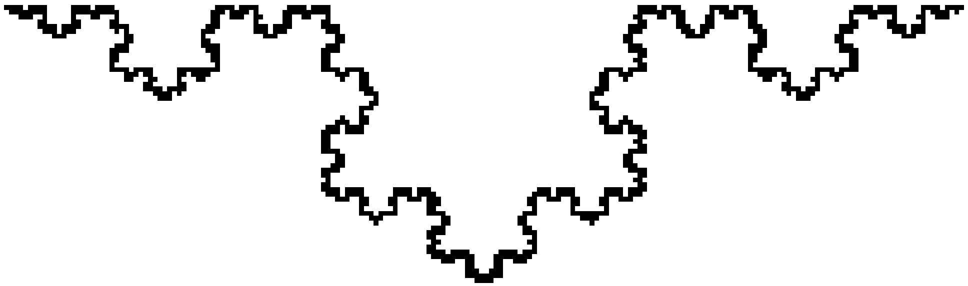
- Fractals: shapes with self-similarity
  - Exact self-similarity



http://en.wikipedia.org/wiki/Self-similarity

Fractals

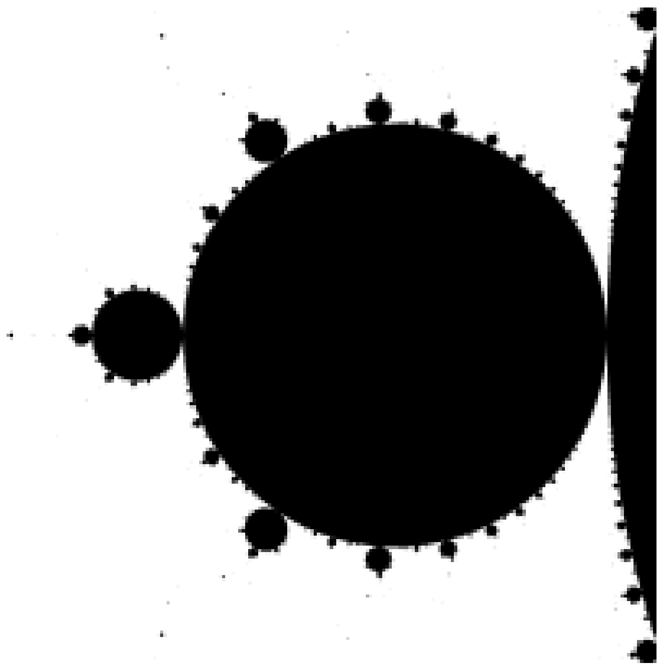
- Fractals: shapes with self-similarity
  - Exact self-similarity



http://en.wikipedia.org/wiki/Self-similarity

Fractals

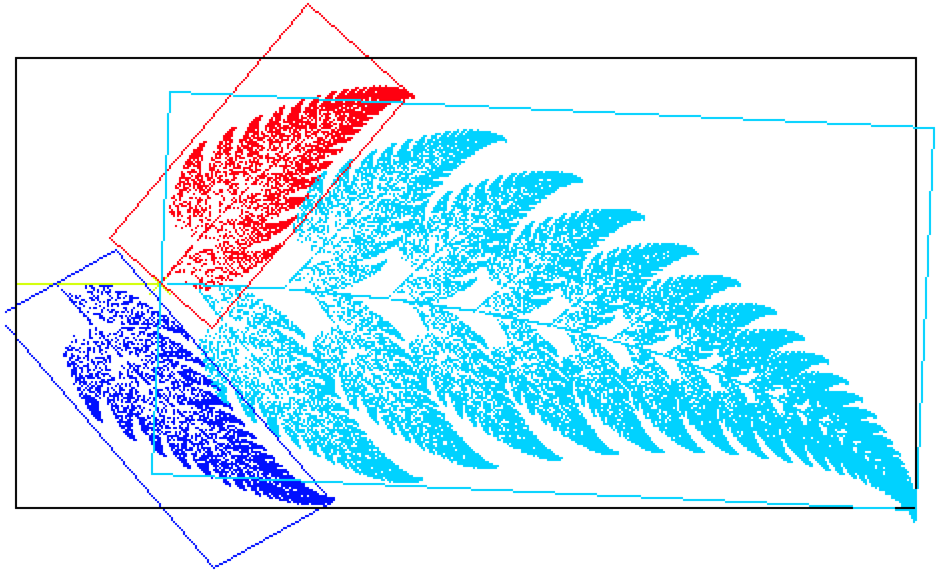
- Fractals: shapes with self-similarity
  - Exact self-similarity



http://en.wikipedia.org/wiki/Self-similarity

Fractals

- Fractals: shapes with self-similarity
  - Exact self-similarity

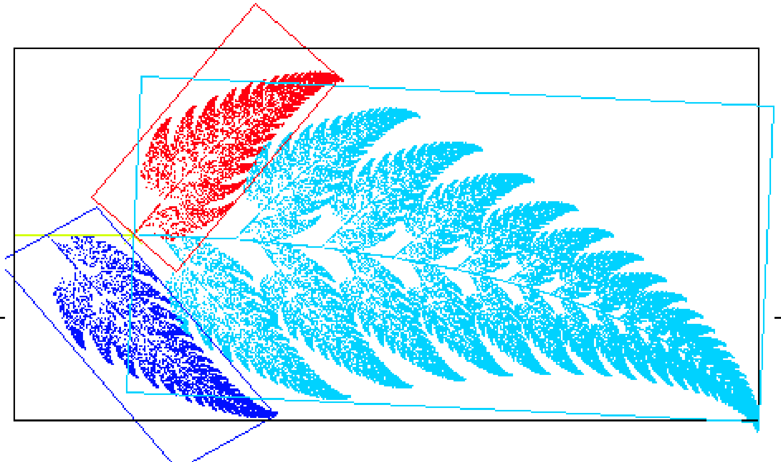


http://en.wikipedia.org/wiki/Self-similarity

IFS fractals

- Consider a point set X.
- Consider a finite set of *contracting* affine transformations $f_1, f_2, \ldots, f_n$.
- We say that X is **self-similar** according to $f_1, f_2, \ldots, f_n$, if

$$X = \bigcup_i f_i(X)$$



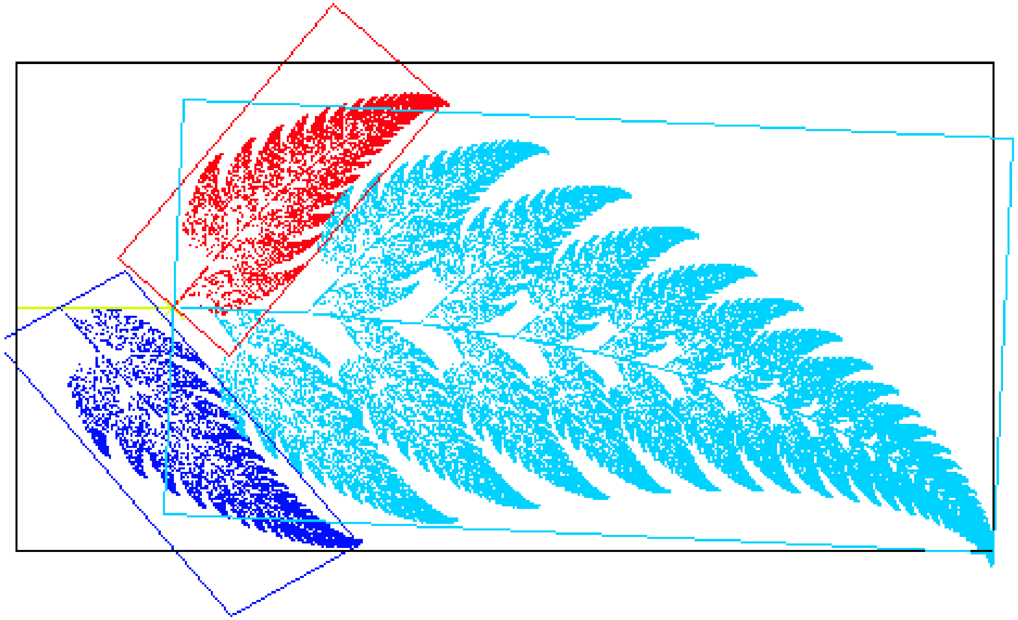
IFS fractals

- It can be shown that for given set of contraction mappings $f_1, f_2, \ldots, f_n$ there exists a *unique* point set that is self-similar according to this function system.

- In simple terms, a given set of self-similarities uniquely defines a particular fractal.

IFS fractals

- This whole fern is specified by providing just four affine transforms (corresponding to red, blue, cyan and yellow rectangles).



Rendering IFS fractals

- Given a set of affine transforms, how to render the corresponding fractal?

Rendering IFS fractals

- Given a set of affine transforms, how to render the corresponding fractal?
  - Method 1: Recursive
  - Method 2: Stochastic

Rendering IFS fractals

- Given a set of affine transforms, how to render the corresponding fractal?
  - **Method 1: Recursive**

```
function render_ifs(level, f_1,f_2,…,f_n) {
   if (level == max) draw_square();
   else
      for (i in 1...n) {
         pushMatrix();
          transform(f_i);
          render_ifs(level+1, f_1,f_2,…,f_n);
         popMatrix();
      }
   }
```

Rendering IFS fractals

- Given a set of affine transforms, how to render the corresponding fractal?
  - **Method 2: Stochastic**

```
function render_ifs(f_1, f_2, ..., f_n) {
    x = (0, 0);
    for (i in 1...num_iterations) {
        r = random(1...n);
        x = f_r(x)
        putpixel(x)
    }
}
```

Quiz

- What does this function draw?

```
function draw_someting() {
    points = {(0, 0), (0.5, 1), (1, 0)};
    x = (0, 0);
    for (i in 1...10000) {
        r = random(0..2);
        x = (x + points[r])/2;
        putpixel(x)
    }
}
```

Quiz

- What does this function draw?

```
function draw_someting() {
    points = {(0, 0), (0.5, 1), (1, 0)};
    x = (0, 0);
    for
        x = (x + points[r])/2;
        putpixel(x)
    }
}
```

See Practice session I

Fractals

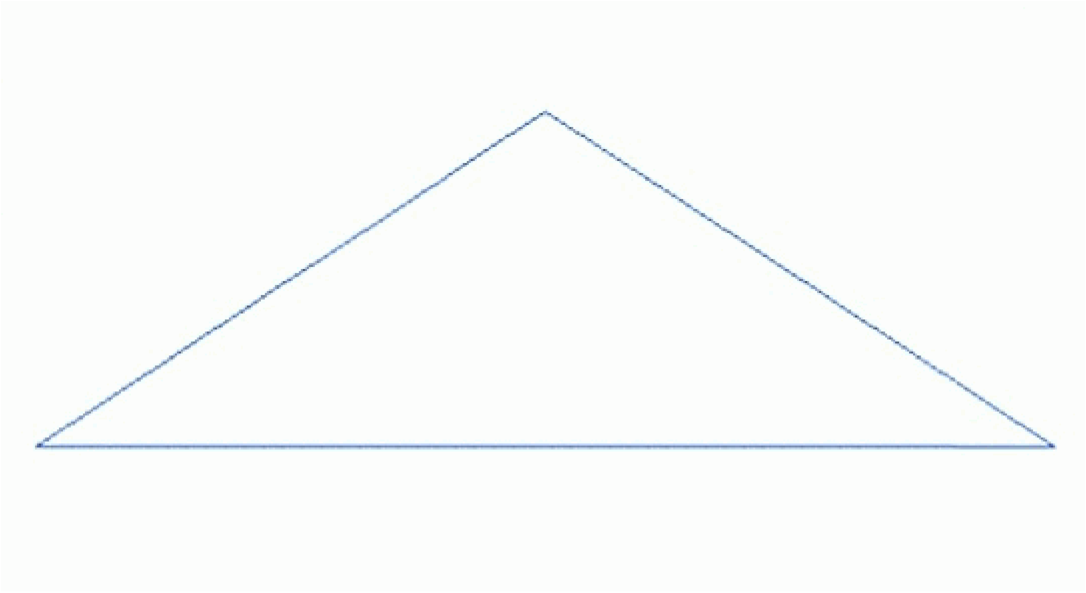
- Fractals: shapes with self-similarity
  - Exact self-similarity
  - Stochastic self-similarity

Fractals

- Fractals: shapes with self-similarity
  - Stochastic self-similarity

Fractal terrains

- Common technique 1: Use Perlin-like noise at several scales (see Lecture 9).
- Common technique 2: Recursive subdivision



Recursive subdivision

Start with a line segment

Recursive subdivision

Divide into two parts at the middle point.
Shift the point by a random amount, typically:

$$\Delta = \mathrm{rand}() \cdot d^a,$$

where a is some constant ≤ 1 and d is segment length.

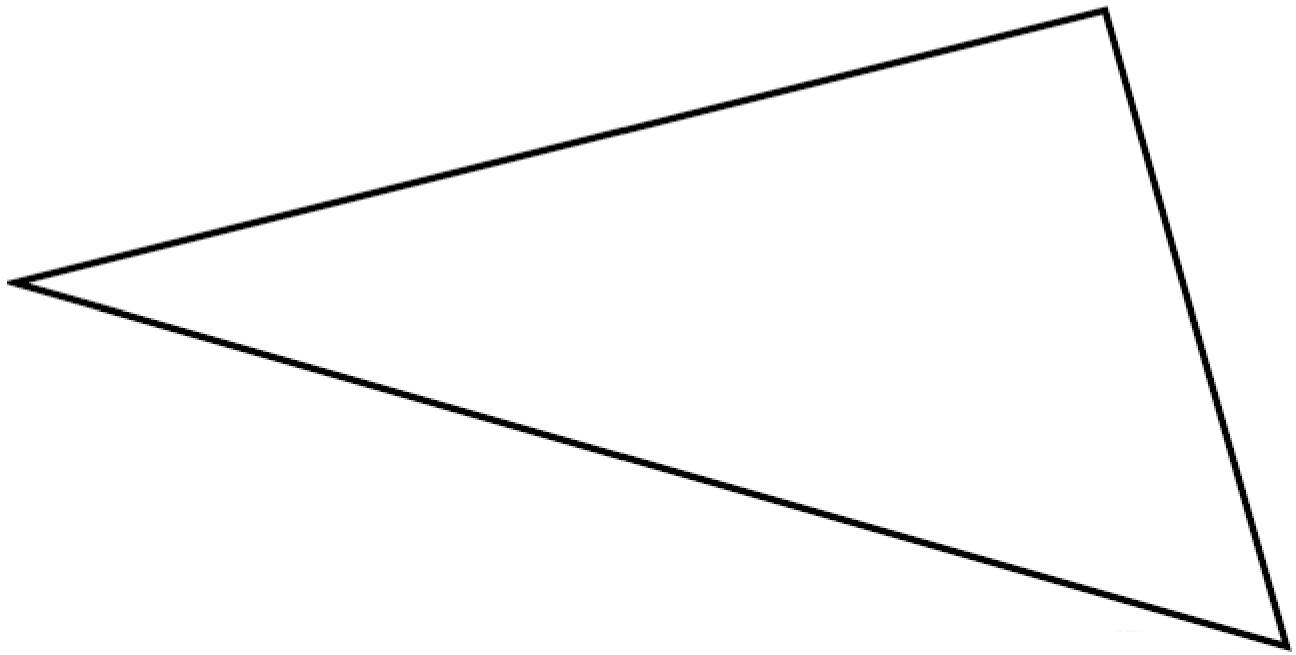
Recursive subdivision

Repeat recursively

Recursive subdivision

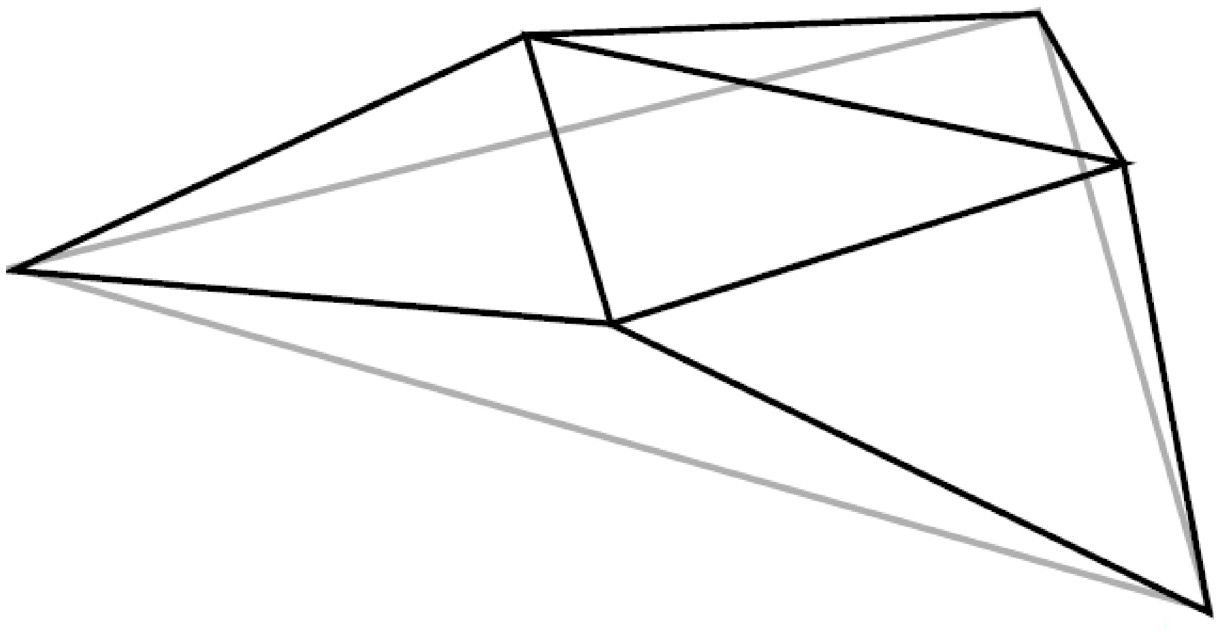
Recursive subdivision

- To subdivide a triangle, simply subdivide each edge and connect them into 4 subtriangles:



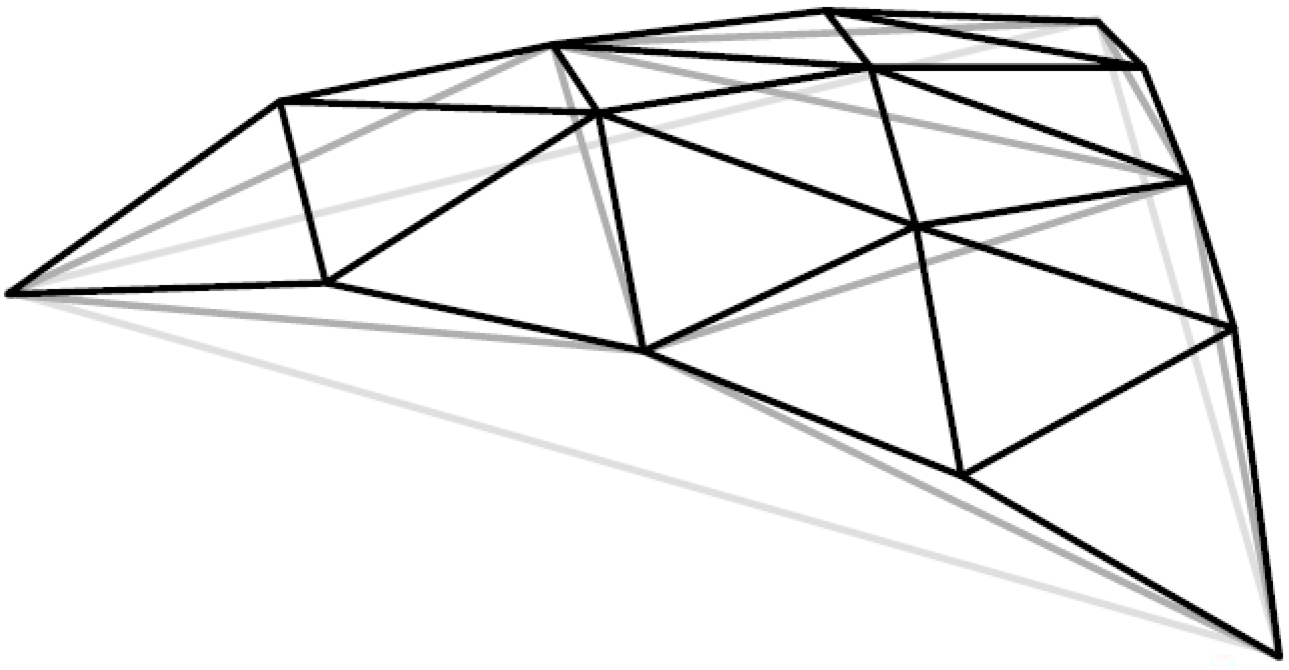
Recursive subdivision

- To subdivide a triangle, simply subdivide each edge and connect them into 4 subtriangles:



Recursive subdivision

- To subdivide a triangle, simply subdivide each edge and connect them into 4 subtriangles:



Recursive subdivision

- Similarly, you can start with a square and subdivide it into 4 subsquares (resulting algorithm is known as “Diamond-square” algorithm)

http://qiao.github.io/fractal-terrain-generator/demo/

http://en.wikipedia.org/wiki/Diamond-square_algorithm

Quiz

- What is the most efficient way to save a randomly generated fractal terrain?

Quiz

- What is the most efficient way to save a randomly generated fractal terrain?

  - It typically suffices to save a single number – the initial seed of the random number generator.

Fractals

- Fractals: shapes with self-similarity
  - Stochastic self-similarity

L-systems (aka generative grammars)

- Consider a set of rules:

  S -> A

  A -> [LA][RA]A

- Start with the symbol "S" and rewrite the string by applying matching rules to symbols:

  S

  => **A**

  => [L**A**][RA]A

  => [L[LA][RA]A][R**A**]A

  => [L[LA][RA]A][R[L**A**][RA]A]A

  => [L[LA][RA]A][R[L[LA][RA]A][RA]A]A

  => ...

L-systems (aka generative grammars)

- Consider a set of rules:

```
S -> A
A -> [LA][RA]A
```

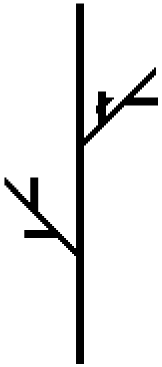
- Now let:
  - `A` denote a piece of code that draws a unit vertical line
  - `[` and `]` denote PushMatrix and PopMatrix
  - `L` denote “translate up by 0.3 and rotate 45 degrees left”
  - `R` denote “translate up by 0.6 and rotate 45 degrees right”

L-systems (aka generative grammars)

- Consider a set of rules:

```
S -> A
A -> [LA][RA]A
```

- Then output `[L[LA][RA]A][R[L[LA][RA]A][RA]A]A` produces



Lindenmayer systems

Some procedural techniques

- Fractals
  - Fractal terrains
  - L-systems (aka Generative grammars)
- Particles
  - Fire, water and magic sparks
  - Boids

Particle systems

- Many effects are best modeled as a combined behavior of a set of “particles”
  - Fire, water, smoke, vapor
  - Grass, hair, fur
  - Explosions, sparks
  - Flocks

Particle system anatomy

- **Emitter**
- **Simulator**
- **Renderer**

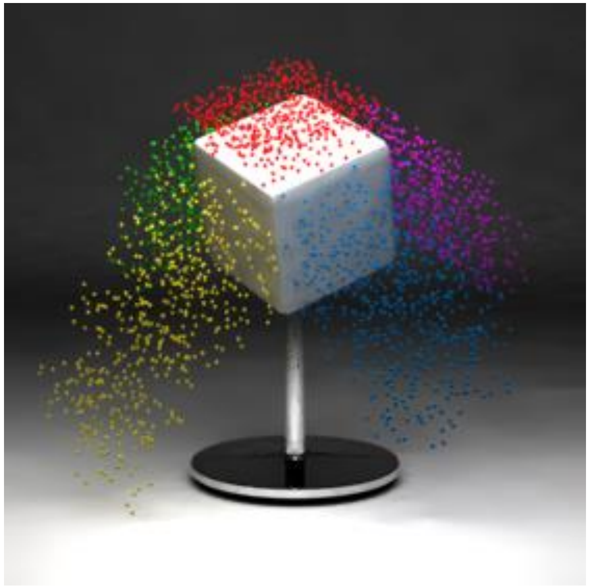
Particle system anatomy

- **Emitter parameters**
  - Where from, at what time and with what parameters are particles emitted?
    - Point emitters
    - Area emitters
    - Volume emitters

    - Emit all at once or gradually?

Particle system anatomy

- **Particle vs “Strand” emitters**
  - For simulating strands of hair, simply render the whole tracks of particles on a single frame



http://en.wikipedia.org/wiki/File:Strand_Emitter.jpg

Particle system anatomy

- **Simulation parameters**
  - What is the lifetime of a particle?
  - What is the behavior of the particle over time? How do its properties change?
    - Physics (gravity, wind, turbulence)?
    - Behaviours?
    - Aging?
    - Particle-particle interactions?
    - Randomness?

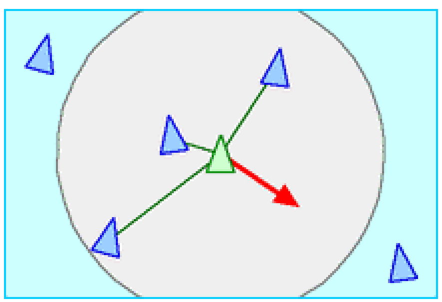
Particle system anatomy

- **Rendering parameters**
  - Render each particle as an image
  - Render each particle as a mesh
  - Use particles to define an implicit surface

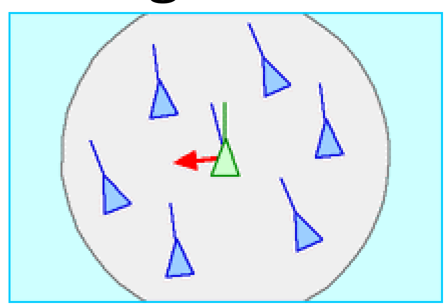
Boids

- A particle system for simulating flocks.
- Each particle follows three simple *steering behaviours:*

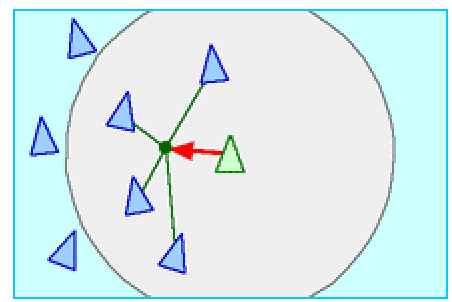
Separation



Alignment



Cohesion



http://www.red3d.com/cwr/boids/

http://www.red3d.com/cwr/steer/

Some procedural techniques

- Fractals
  - Fractal terrains
  - L-systems (aka Generative grammars)
- Particles
  - Fire, water and magic sparks
  - Boids

Course wrap-up

Modeling | Rendering | Animation

Course wrap-up

Modeling	Rendering	Animation

Scene graph

Model-View-Projection Transforms

- Meshes
- Point clouds
- Implicit surfaces
- Distance fields
- Parametric curves & surfaces

- Fractals, L-systems
- Particle systems

- Color spaces
- Texture maps

- **Lighting models**

Course wrap-up

Modeling

Scene graph

Model-View-Projection Transforms

- Meshes
- Point clouds
- Implicit surfaces
- Distance fields
- Parametric curves & surfaces

- Fractals, L-systems
- Particle systems

- Color spaces
- Texture maps

- **Lighting models**

Rendering

2D graphics: lines & triangles

Standard pipeline

- Shaders
- Hidden surfaces
- Shadows

- **Raycasting**
- Raytracing
- Raymarching

- **Rendering equation**
- Radiosity
- Path tracing

Sampling & antialiasing

Animation

Course wrap-up

Modeling

Scene graph

Model-View-Projection Transforms

- Meshes
- Point clouds
- Implicit surfaces
- Distance fields
- Parametric curves & surfaces

- Fractals, L-systems
- Particle systems

- Color spaces
- Texture maps

- **Lighting models**

Rendering

2D graphics:
lines & triangles

Standard pipeline

- Shaders
- Hidden surfaces
- Shadows

- **Raycasting**
- Raytracing
- Raymarching

- **Rendering equation**
- Radiosity
- Path tracing

Sampling & antialiasing

Animation

GUI application,
Event loop,
Model-view-controller

Keyframe animation,
Skeletal animation

Physics simulation

Particle systems, Boids

Parametric curves

C/C++ | **Allegro** | **OpenGL** | **GLSL** | **OGRE** | **Blender** | **Unity**

Modeling

Scene graph

Model-View-Projection Transforms

- Meshes
- Point clouds
- Implicit surfaces
- Distance fields
- Parametric curves & surfaces

- Fractals, L-systems
- Particle systems

- Color spaces
- Texture maps

- **Lighting models**

Rendering

2D graphics: lines & triangles

Standard pipeline

- Shaders
- Hidden surfaces
- Shadows

- **Raycasting**
- Raytracing
- Raymarching

- **Rendering equation**
- Radiosity
- Path tracing

Sampling & antialiasing

Animation

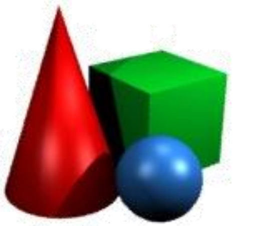
GUI application, Event loop, Model-view-controller

Keyframe animation, Skeletal animation

Physics simulation

Particle systems, Boids

Parametric curves



Next steps

- There’s much more on **modeling**, e.g.:
  - Mesh segmentation, tessellation, simplification,
  - Various procedural techniques (terrains, fractals, …)
  - Other lighting models & texture mapping techniques, …
- More on **rendering**:
  - Better or faster rendering equation solvers
  - More detailed light physics
  - Non-photorealistic rendering & special FX, …
- More on **animation**:
  - Fluid & air simulation, inverse kinematics & constraint solving, motion tracking, …

C/C++ | Allegro | OpenGL | GLSL | OGRE | Blender | Unity

Modeling

Scene graph

Model-View-Projection Transforms

- Meshes
- Point clou
- Implicit s
- Distance
- Parametric curves & surfaces

- Fractals, L-systems
- Particle systems

- Color spaces
- Texture maps

- **Lighting models**

Rendering

2D graphics:
lines & triangles

Standard pipeline

- **Raycasting**
- Raytracing
- Raymarching

- **Rendering equation**
- Radiosity
- Path tracing

Sampling & antialiasing

Animation

GUI application,
Event loop,
Model-view-controller

me animation,
tal animation

cs simulation

Particle systems, Boids

Parametric curves

Thank you!