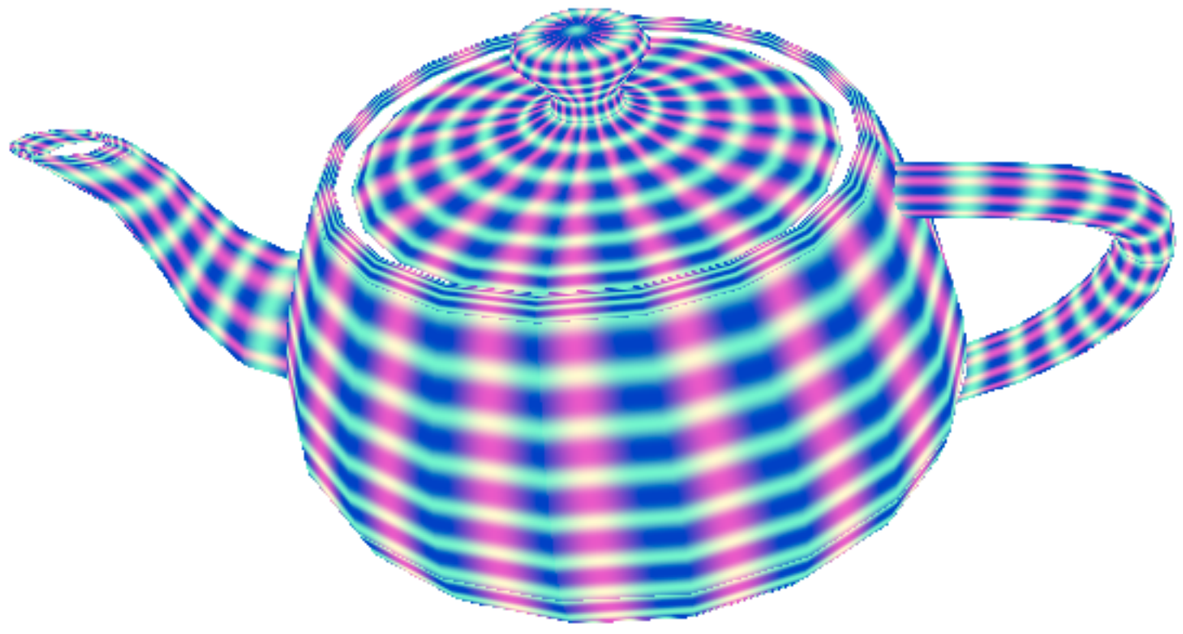
Information Coding / Computer Graphics, ISY, LiTH

TNM084
Procedural images

Ingemar Ragnemalm, ISY



Lecture 6

Procedural geometry

Fractals

Procedural geometry

Producing models with code:

Simple shapes with customizable detail

Shapes with repeating detail

Sampling functions

Sweeping

Tools for geometry generation

Simple shapes with customizable detail

Example: Sphere

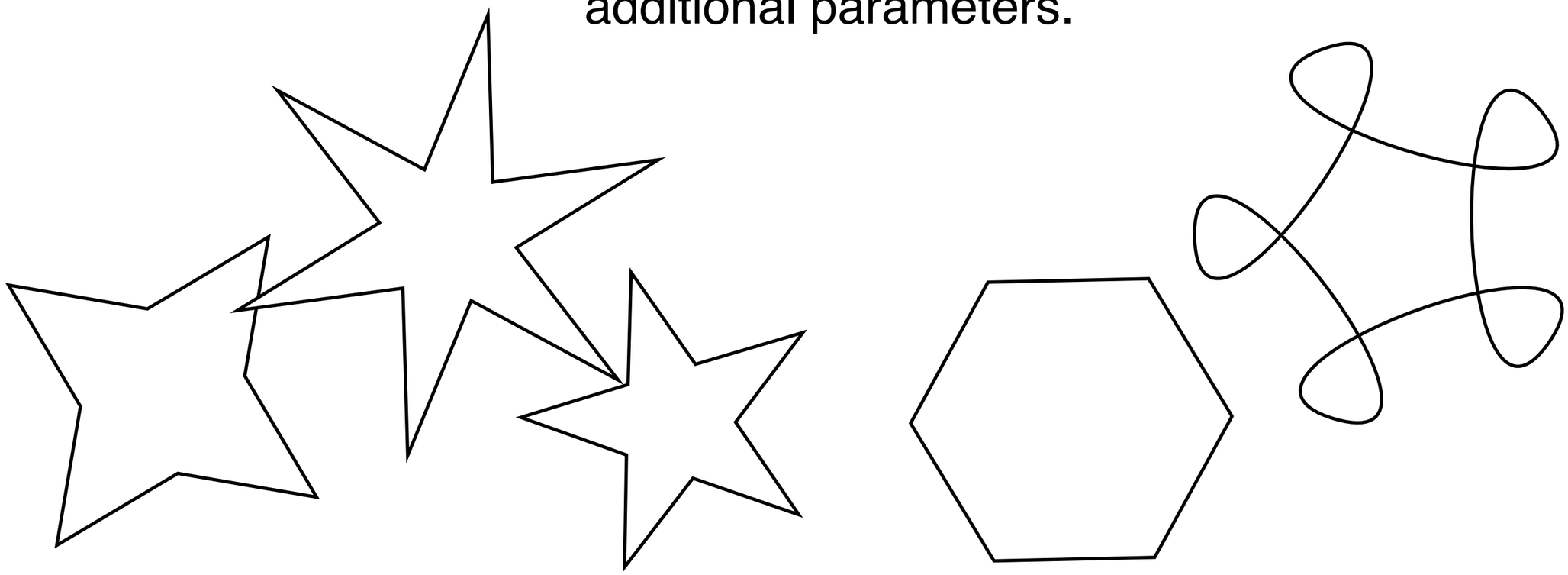
1. Define in polar coordinates. Setting: Number of steps along each coordinate

2. Define by tesselation. Setting: Number of tesselation steps

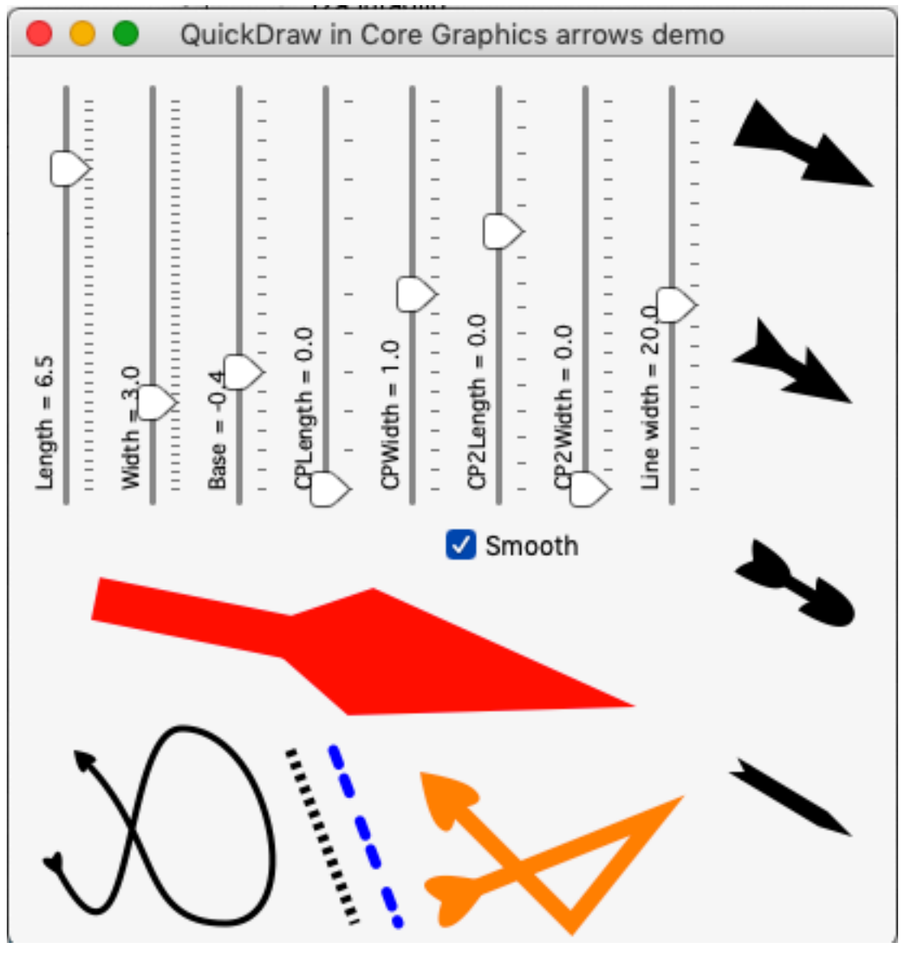
Shapes with repeating detail

Examples: Stars, polygons

Number of details can be customizable but also additional parameters.



Another example: Arrows



No repeating detail but highly customizable.

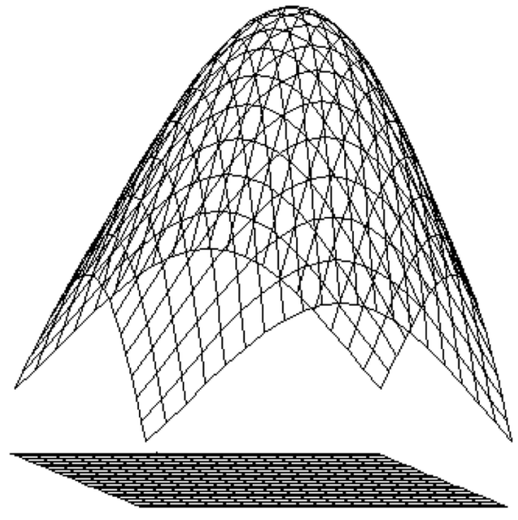
All arrows are made from code, the same code!

Sampling a function

A model can be constructed from a mathematical function

Sample at suitable distance and create polygons (triangles)

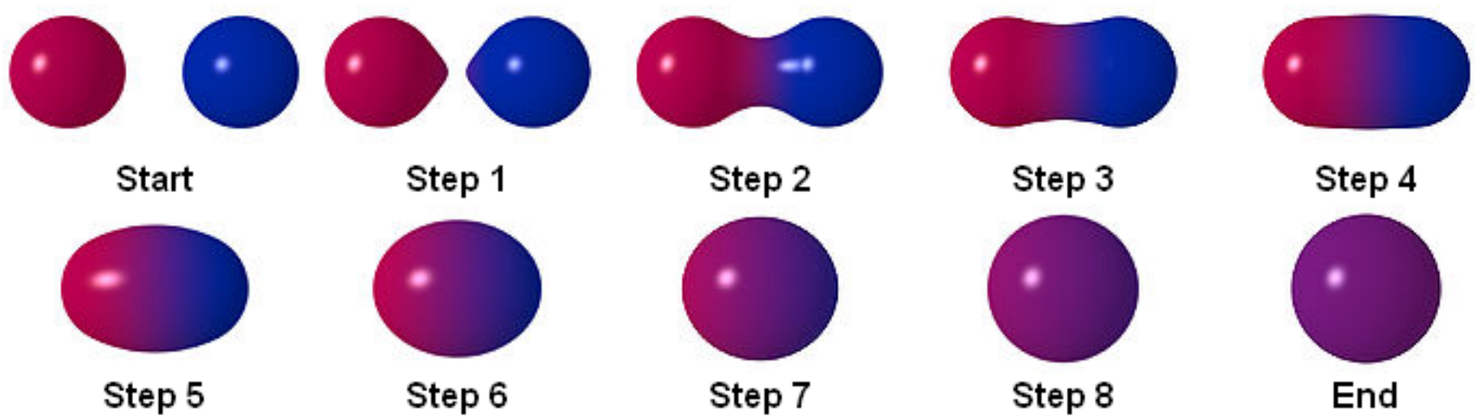
Find functions that create closed models.



Blobby objects, Metaballs

Build shapes from sets of points. Represent surfaces by distance functions

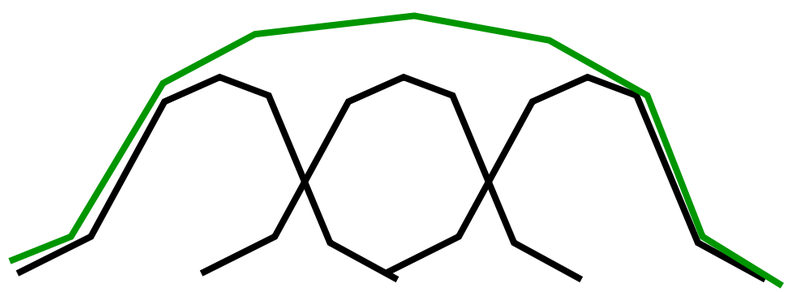
A shape = set of points + weights



Yet another image stolen from Wikipedia

Blobby objects

Gaussian functions (”Gaussian bumps”)



$$f(x) = \sum b_k * \exp(x - p_k)$$

Surface at f(x) = threshold

But Gaussians are not fast!

Blobby objects

Gaussian: A function that falls off towards zero and has a top at 1 and!

Can we use something else? Best if:

- Smooth
- Reaches zero at a known radius! (Why?)

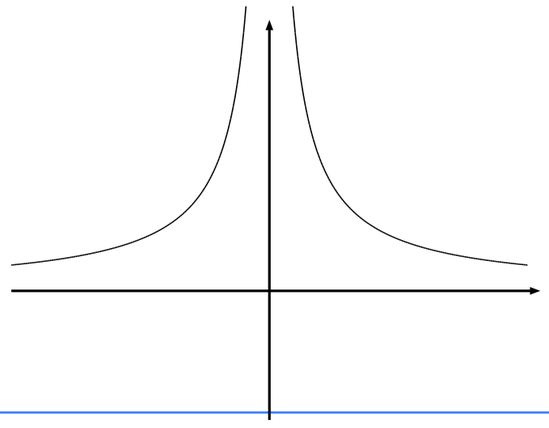
Possible function:

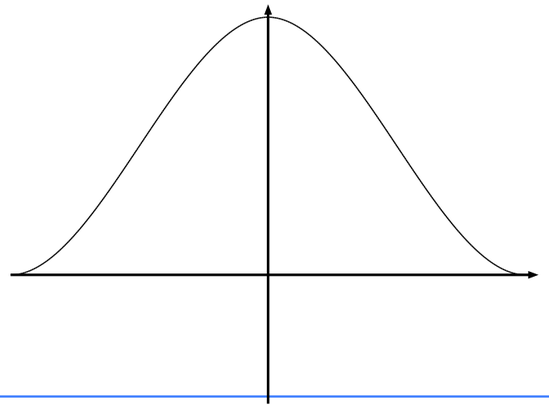
$$f(x) = R / \mathrm{sqrt}((x - x_0)^2)$$

Why can this work? Big peak in the middle, that is not a gaussian!

Even better if we can avoid division and square root.

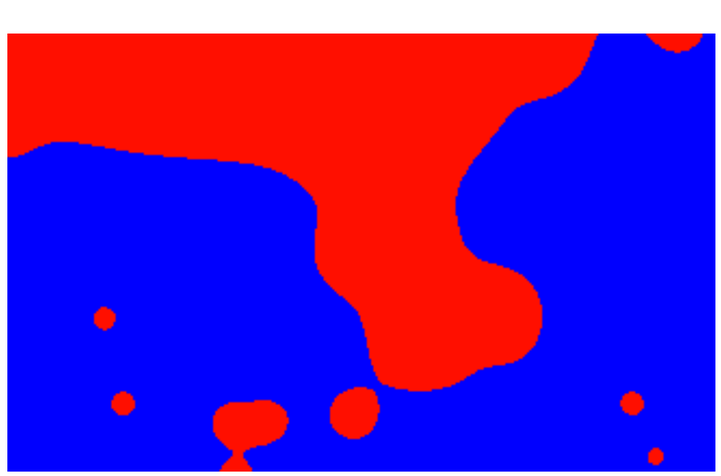
Smoothstep is perfectly feasible and a decent approximation of a gaussian.





Drawing blobby objects

In 2D, blobby objects are easily made by just checking if a pixel is inside/outside.



In 3D, we want to produce surfaces! How can we do that?

Marching cubes

Voxels with density values
Treshold in density
Voxels are corners of cubes
Create polygons in the treshold.

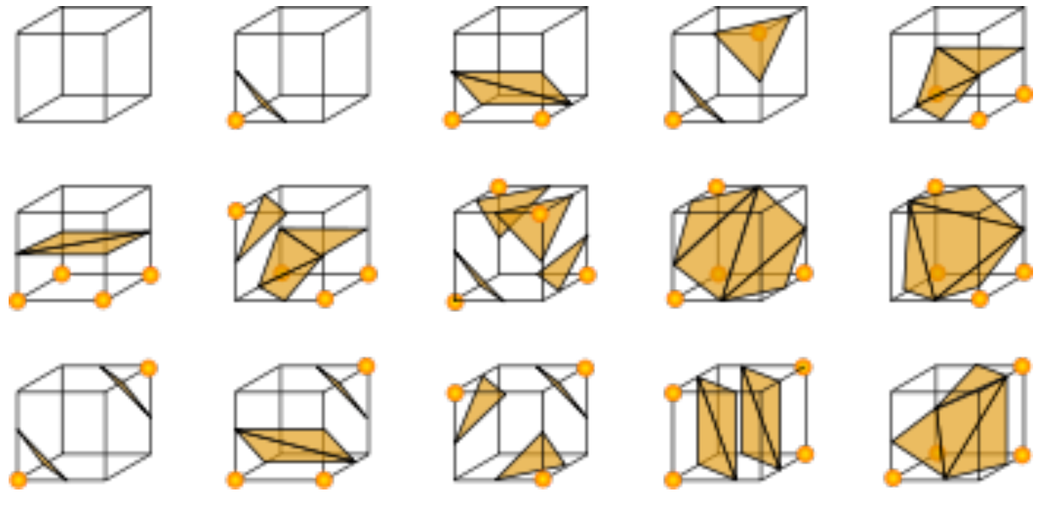
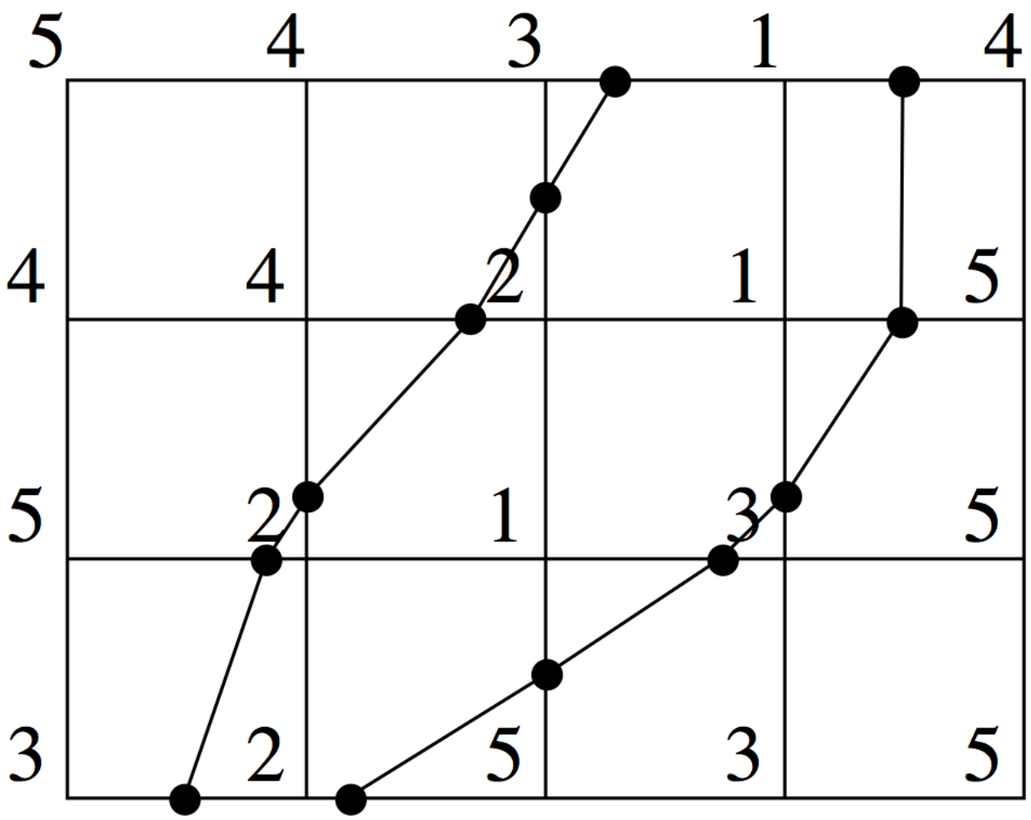


Bild fr Wikipedia

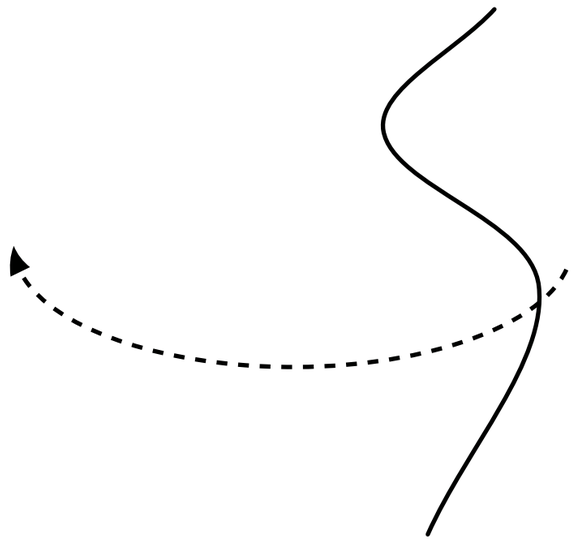
Marching squares - Marching cubes i 2D

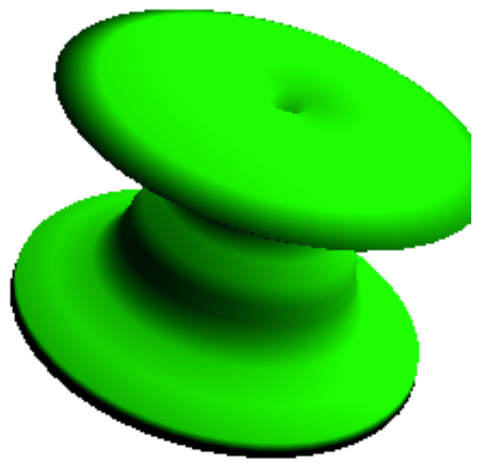


Sweeping (Rotational sweep)

Circular symmetric shapes

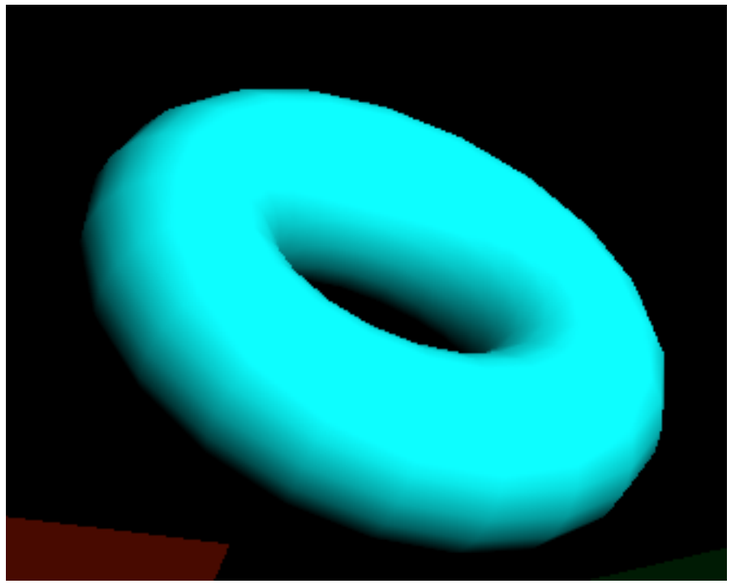
Define by a curve (preferrably a spline) which is rotated and vertices created along it.





Sweeping of 2D shapes

Sweeping can also be made with closed curves. Typical case: The torus, sweeping of a circle.



Normal vectors

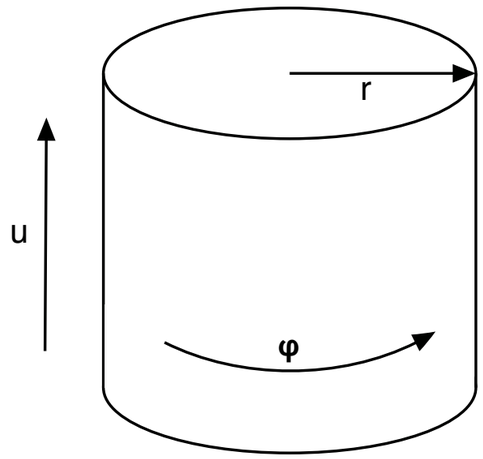
Hard problem for many procedural shapes

- Calculate from known geometry. Easy for some shapes, but not all.
- Post-processing. Find polygons (e.g. triangles) touching each vertex. Caclulate normal.

Analytical normal vectors for sweeping

Variation along height and around object

Simple case: Cylinder



$$p(u, \varphi) = (r\sin\varphi, u, r\cos\varphi)$$

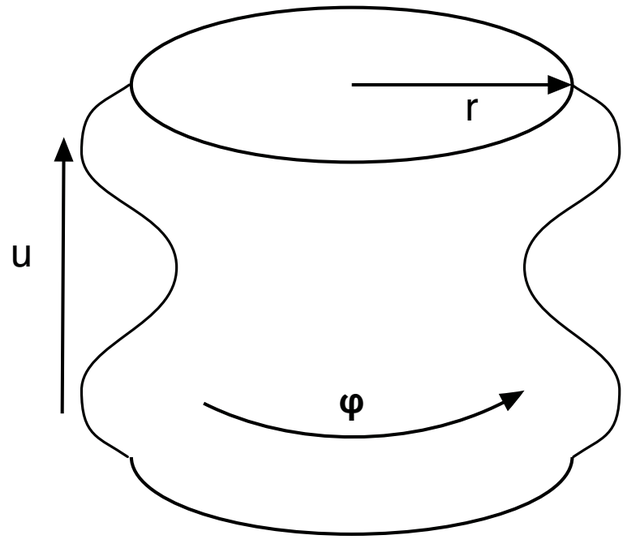
$$\frac{dp}{du} = (0, 1, 0)$$

$$\frac{dp}{d\varphi} = (r\cos\varphi, 0, -r\sin\varphi)$$

$$n(u, \varphi) = \frac{dp}{du} \times \frac{dp}{d\varphi} = (r\sin\varphi, 0, r\cos\varphi)$$

As expected: Straight out

Harder case: Sin wave



$$p(u, \varphi) = (\sin\varphi(\sin u + r), u, \cos\varphi(\sin u + r))$$

$$\frac{dp}{du} = (\sin\varphi\cos u, 1, -\sin\varphi\cos u)$$

$$\frac{dp}{d\varphi} = (\cos\varphi(\sin u + r), 0, -\sin\varphi(\sin u + r))$$

$$n(u, \varphi) = \frac{dp}{du} \times \frac{dp}{d\varphi} = (\sin\varphi, (\cos^2\varphi - \sin^2\varphi)\cos u, \cos\varphi)$$

Normal vector varies by height

Calculating normal vectors from geometry

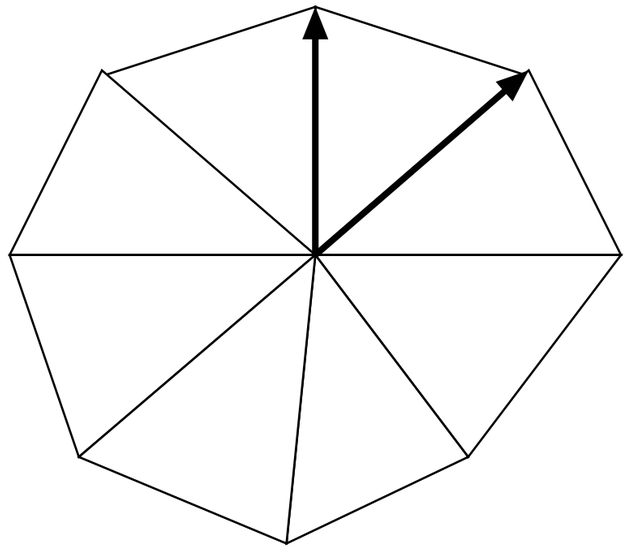
- Find all neighbor triangles. Calculate normals for each triangle with cross product. Make a weighted average.

- Triangle method: Find three vertices enclosing the vertex. Find normal by cross product of two edges.

- Cross method. Find four vertices enclosing the vertex. Make a vector from each opposing pair, get normal with cross product.

All triangles method

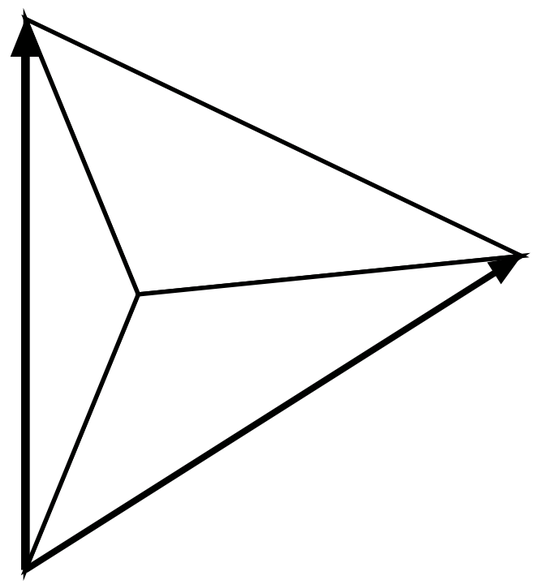
Most precise and most general. Also most work.



Triangle method

Only three vertices. Surprisingly good.

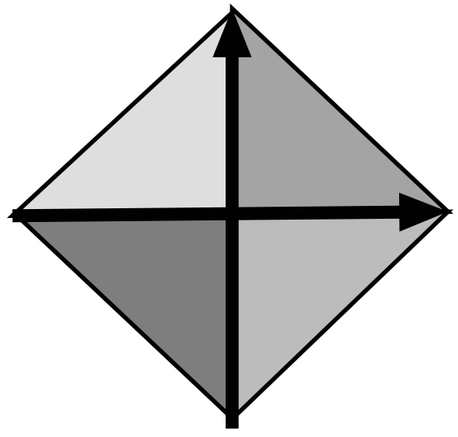
Note that the target vertex is not involved. How can that work?

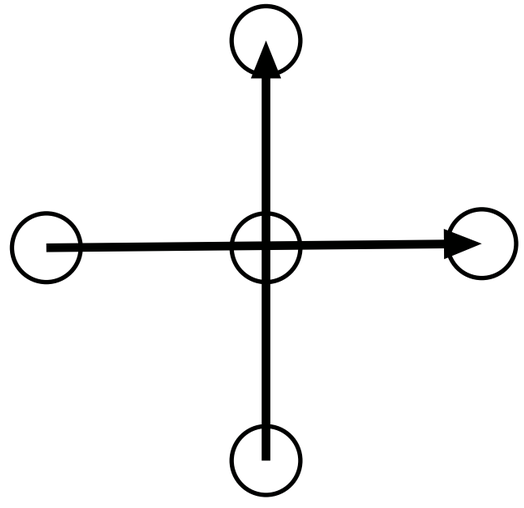


Cross method

Four vertices. Good if vertex locations are (more or less) axis aligned.

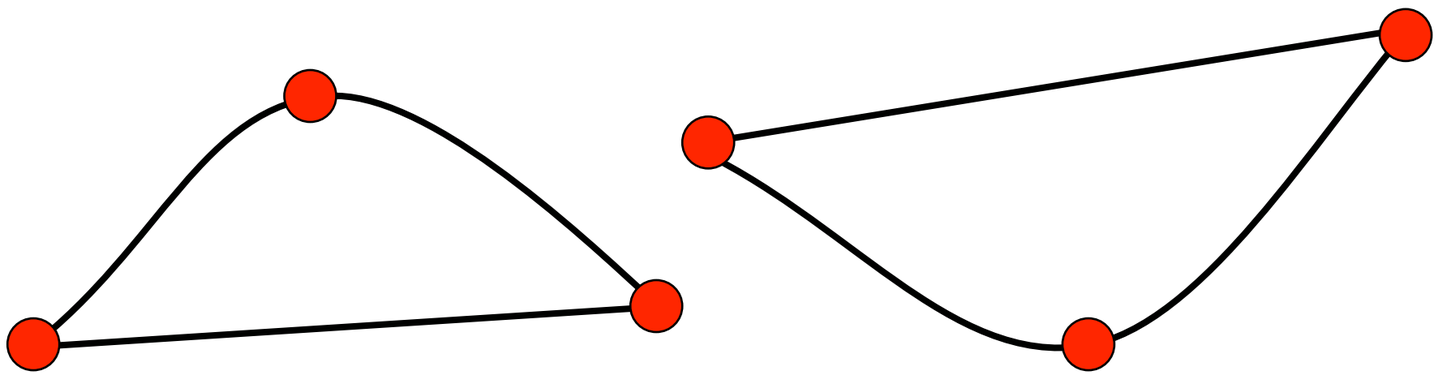
Again, the target vertex is not involved.





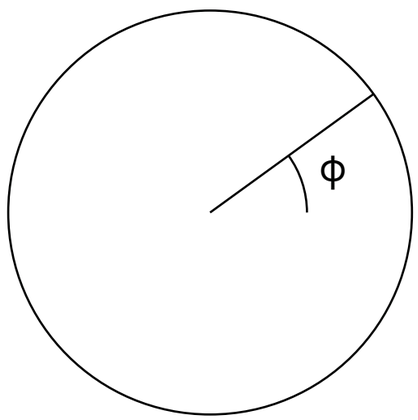
Why can we get a good approximation from the neighbors alone?

The neighbors form a plane for which the normal tends to be a good normal for the vertex inside the triangle/cross. It doesn't matter hos the center vertex moves.

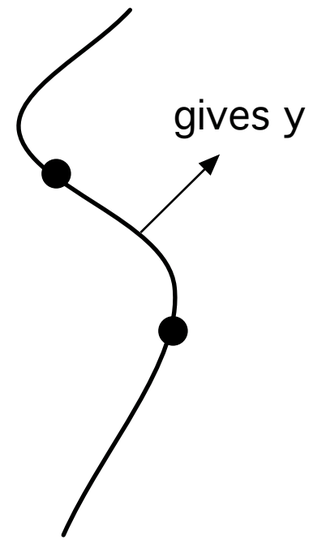


Sweeping and normals

Simple case for normal vectors. Only the Y component varies along a vertical slice, the others are given by the rotation step. Thus, just taking one step up and down along the spline suffices!, Or use the mathematical derivative of the spline like above.



cos ϕ, sin ϕ
give x and z



Turtle graphics

"Pen" in relative (local) coordinates.

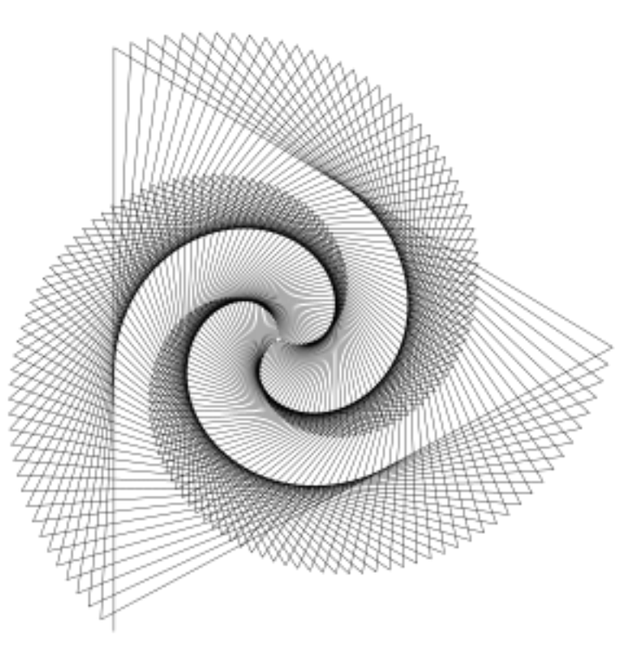
Pen position and orientation

Simple commands, "forward", "turn 90 degrees"

2D or 3D system.

Turtle graphics examples

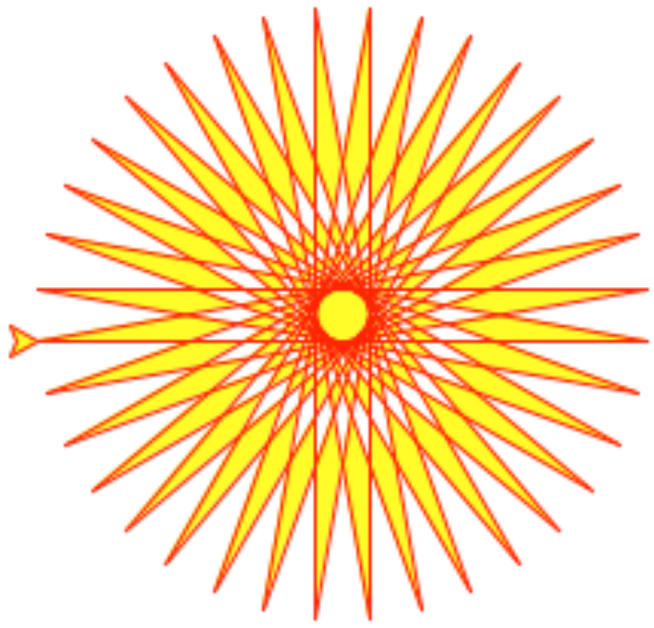
From Wikipedia



Turtle graphics examples

From Python lib/turtle.py

```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```



OpenSCAD, the procedural geometry CAD program

A CAD modelling software based on *code*! You are literally programming your models, and get a 3D model out.

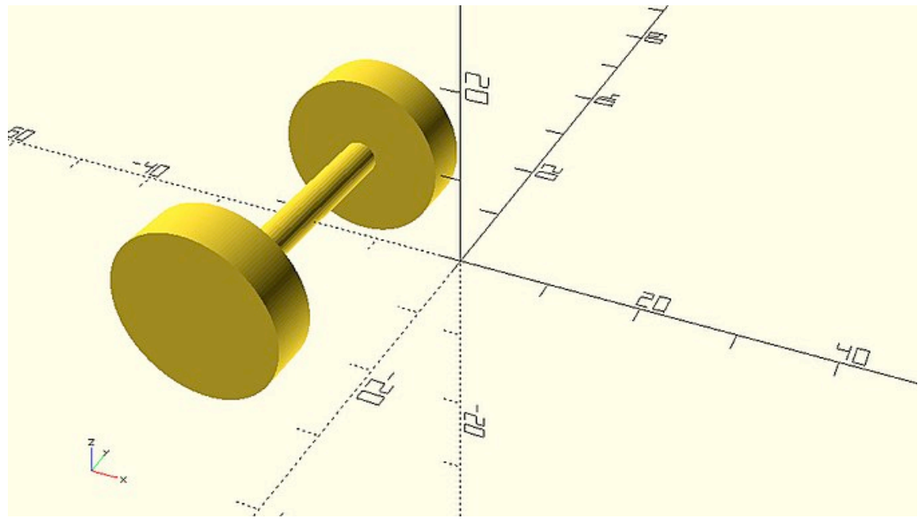
Simple example just put together existing shapes.

Advanced use model with coordinates and splines.

OpenSCAD wheels, transformations

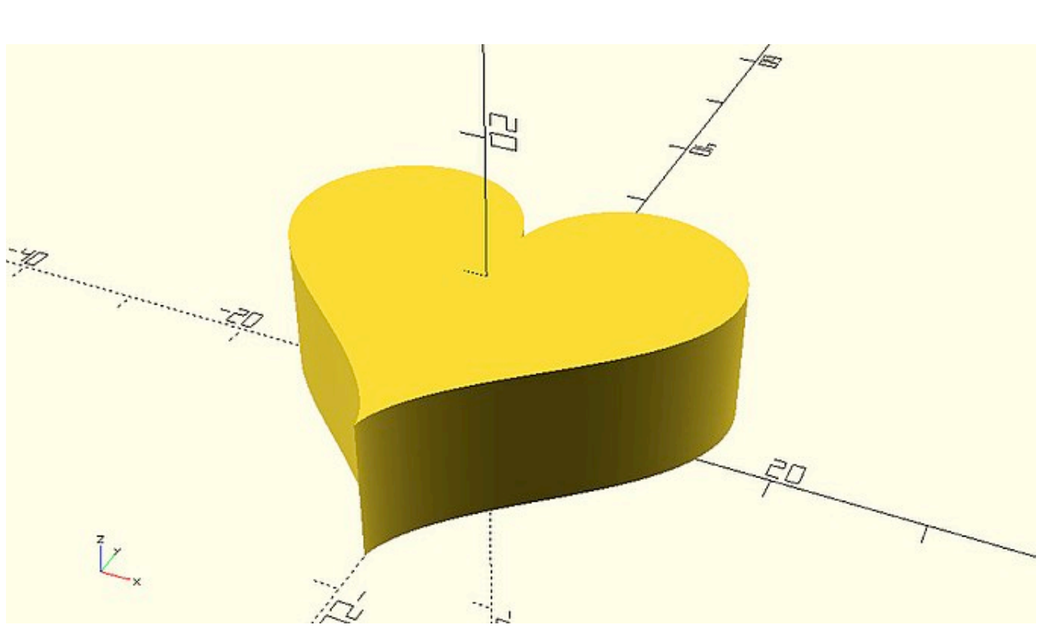
Uses existing parts

```
use <vehicle_parts.scad>;
$fa = 1;
$fs = 0.4;

wheelbase = 40;
track = 35;

translate([-wheelbase/2, track/2])
    simple_wheel();
translate([-wheelbase/2, -track/2])
    simple_wheel();
translate([-wheelbase/2, 0, 0])
    axle(track=track);
```



OpenSCAD heart example

points = [for (t=[0:step:359.999]) [16*pow(sin(t),3), 13*cos(t) - 5*cos(2*t) - 2*cos(3*t) - cos(4*t)]];



Old OpenGL procedural geometry

Big thing in old OpenGL

Used the "immediate mode"

Recorded to "display lists" for performance

Phased out with OpenGL 3.

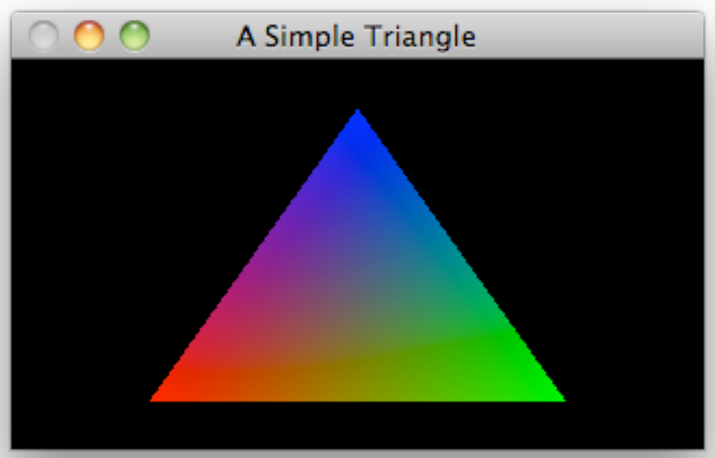
Immediate mode

Specify geometry by function calls per vertex:

```
glBegin(GL_POLYGON);
    glColor3f(1, 0, 0); glVertex3f(-0.6, -0.75, 0.5);
    glColor3f(0, 1, 0); glVertex3f(0.6, -0.75, 0);
    glColor3f(0, 0, 1); glVertex3f(0, 0.75, 0);
glEnd();
```

draws a triangle with different colors.

Too many function calls for large models.

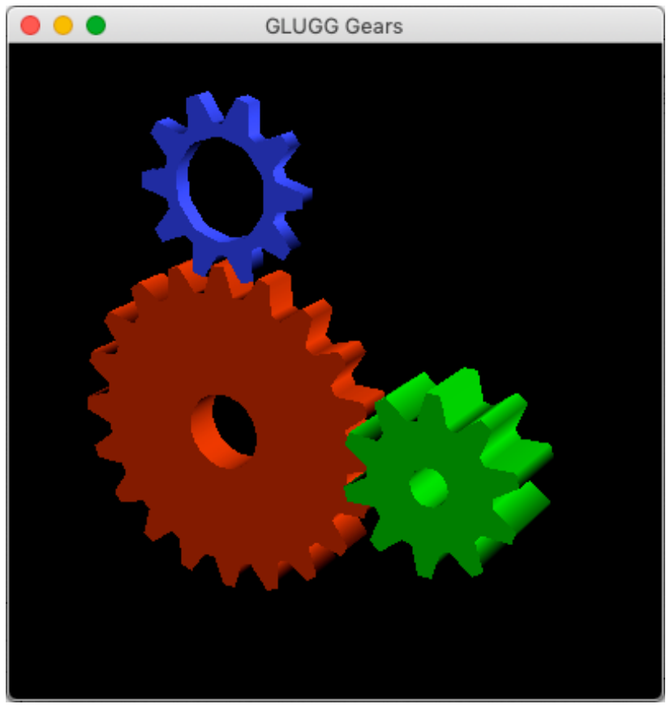


https://cs.lmu.edu/~ray/notes/openglexamples/

Classic examples

Gears (glxGears standard example in Mesa)

Creates three matching gears procedurally.



The matrix stack

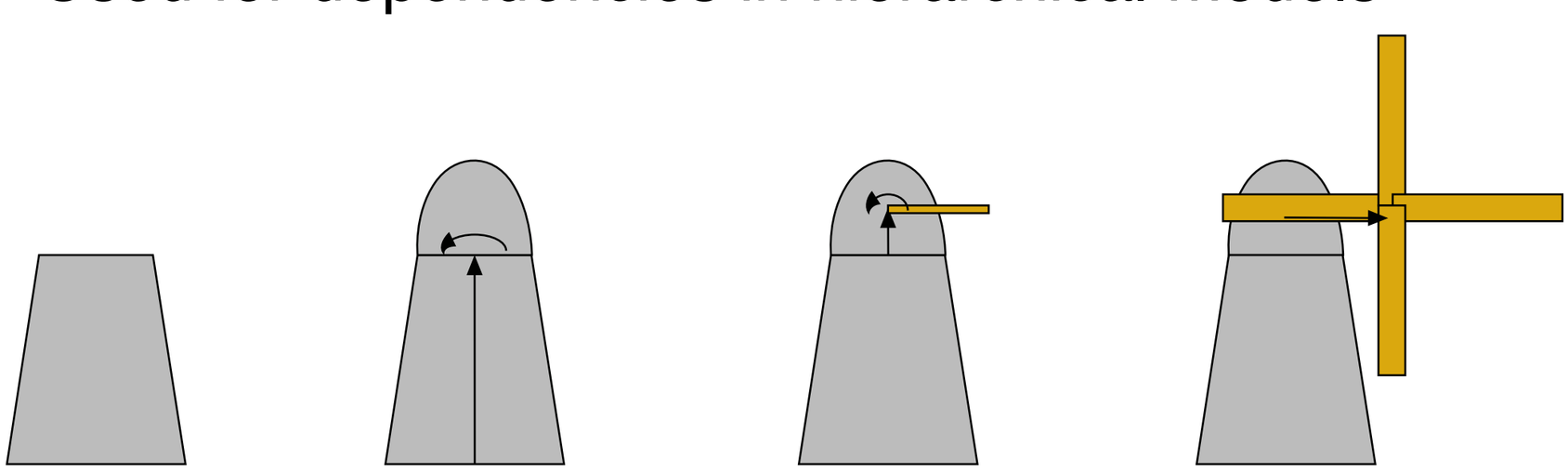
Old OpenGL managed a "matrix stack" for dealing with hierarcical models.

glPushMatrix()
glPopMatrix()

Makes it easier to build hierarchical models

Revisited: Transformations to sub-systems under model coordinates

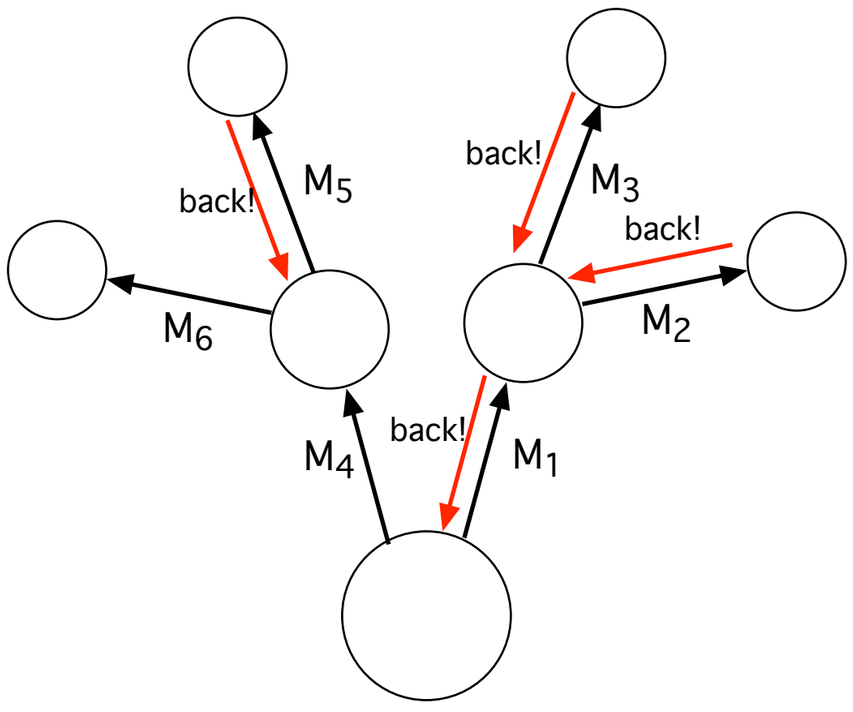
Used for dependencies in hierarchical models



Model coordinates	Top of windmill	Axis for blades	Blade
Body of windmill	with rotation for top (around y)	The axis can rotate (around x or z)	Blades rotate by following the rotation of the axis

What if you have multiple braches/ dependencies?

Use the matrix stack to get back to earlier nodes



For every node that you want to go back to, save the current matrix with glPushMatrix().

Go back with glPopMatrix()

Why do we care?

Procedural modelling in OpenGL has fallen out of fashion -
but is it useless?

How about doing this with modern code?

OpenGL Utilities for Geometry Generation (GLUGG)

My code package for creating procedural geometry in a modern way, but similar to the old way.

- Create geometry with similar calls, gluggVertex etc
- Supports transformations and a matrix stack.
- Generates a vertex/polygon list (optionally with an index array) for uploading to a VAO.

Intended for generating geometry at the initialization of a program.

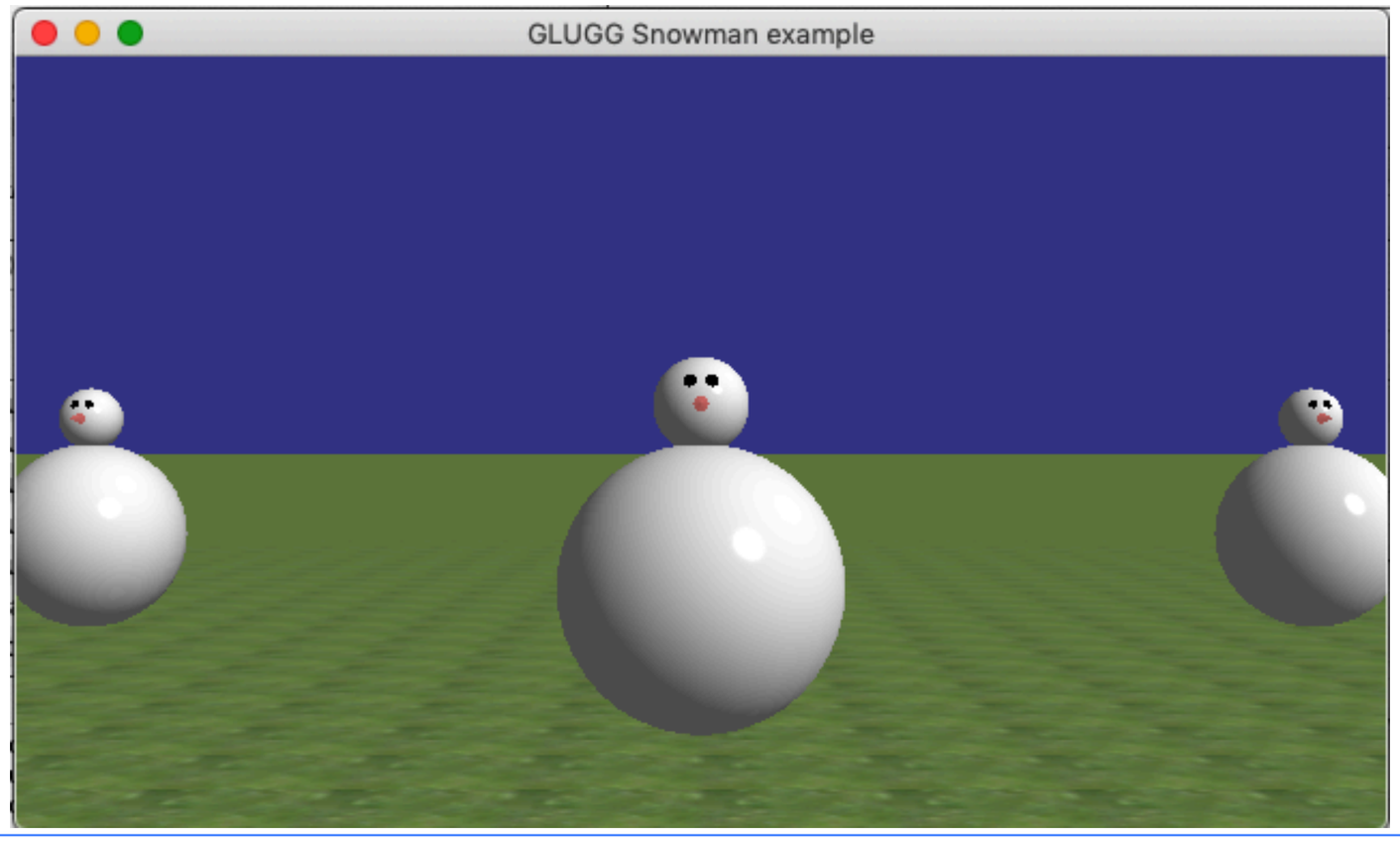
Simple usage of GLUGG: Triangle

```
#include "MicroGlut.h"
#include "GL_utilities.h"
#include "glugg.h"

GLuint program; // Shader
gluggModel triangle;

void draw(void)
{
   glClear(GL_COLOR_BUFFER_BIT);

   gluggDrawModel(triangle, program);

   glutSwapBuffers();
}
```

```
void init(void)
{
   program = loadShaders("minimal.vert", "minimal.frag");

   /* make the triangle */
   gluggBegin(GLUGG_TRIANGLES);
   gluggVertex(-0.5,-0.5,0);
   gluggVertex(0.5,-0.5,0);
   gluggVertex(-0.5,0.5,0);
   triangle = gluggBuildModel(0);
}

int main(int argc, char *argv[])
{
   glutInit(&argc, argv);

   glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
   glutInitContextVersion(3, 2);

   glutCreateWindow("GLUGG White Triangle");
   init();
   glutDisplayFunc(draw);

   glutMainLoop();
   return 0;
}
```

Building with multiple parts and the matrix stack

The snowman (based on old demo from lighthouse3d)



The code for making the snowman

Parts made in other functions are built together using transformations and the matrix stack

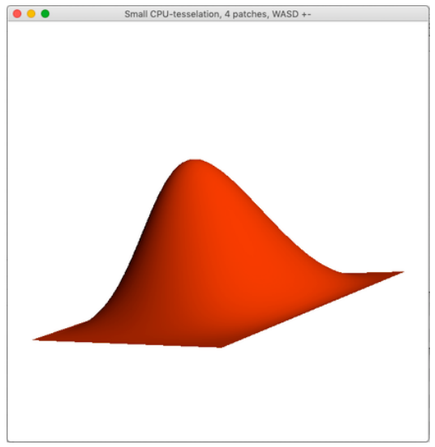
```
gluggModel MakeSnowman()
{
   gluggBegin(GLUGG_TRIANGLES);

   gluggColor(1.0f, 1.0f, 1.0f);

// Draw Body
   gluggTranslate(0.0f ,0.75f, 0.0f);
   gluggSphere(20,20, 0.75f);

// Draw Head
   gluggTranslate(0.0f, 0.95f, 0.0f);
   gluggSphere(20,20, 0.25f);

// Draw Eyes
   gluggPushMatrix();
   gluggColor(0.0f,0.0f,0.0f);
   gluggTranslate(0.05f, 0.10f, 0.18f);
   gluggSphere(10,10, 0.05f);
   gluggTranslate(-0.1f, 0.0f, 0.0f);
   gluggSphere(10,10, 0.05f);
   gluggPopMatrix();

// Draw Nose
   gluggColor(1.0f, 0.5f , 0.5f);
   gluggRotate(M_PI/2.0,1.0f, 0.0f, 0.0f);
   gluggCone(10, 0.5, 0.08f);

   return gluggBuildModel(0);
}
```

Building with Bézier surfaces

Bézier surfaces are built-in!

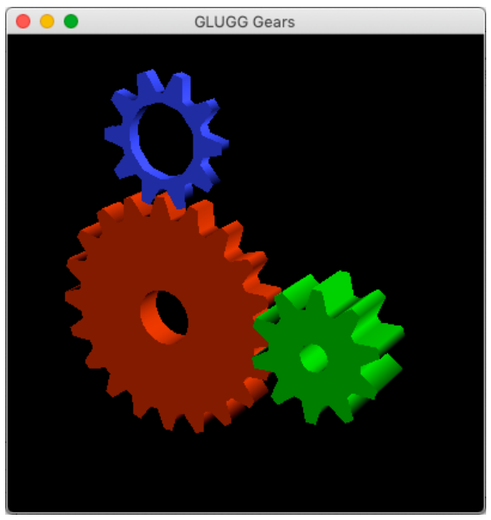
G^1/C^1 continuity is up to you (currently).

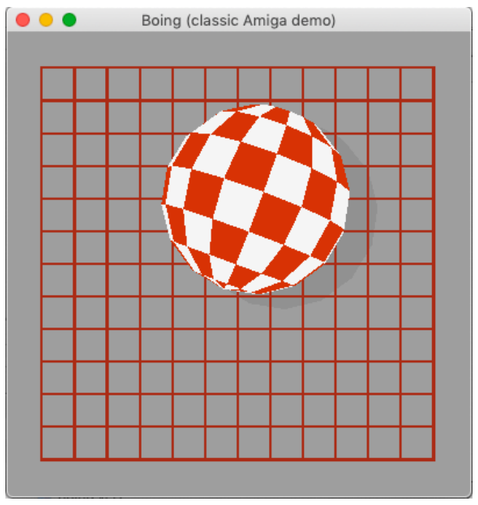
Used for the 4-patch "surface" and for the Utah Teaset!

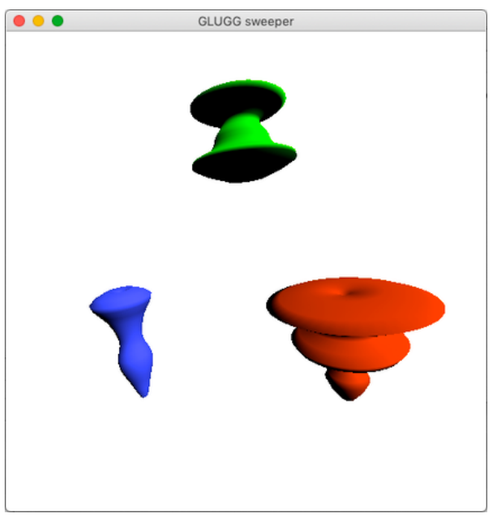


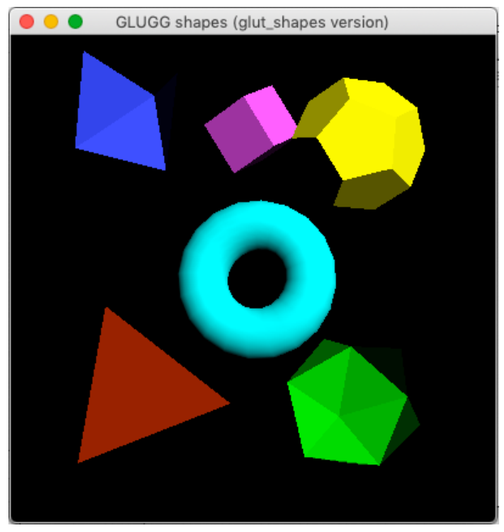
Plenty of demos and a (hopefully) decent documentation

…but very little used by others than myself - until now.









Usage in lab 3

GLUGG will be used in the first part of Lab 3. It was used for that last year with old results.

The latest version can be found here:

https://computer-graphics.se/packages/glugg.html

Procedural geometry, summary (so far)

Produces flexible, customizable and detailed geometry

Sweeping, normal vectors

Matrix stack for handling multiple branches

Several approaches:

Turtle graphics
OpenSCAD
Old-style OpenGL
GLUGG

But where does the noise/randomness come in?

Right now: