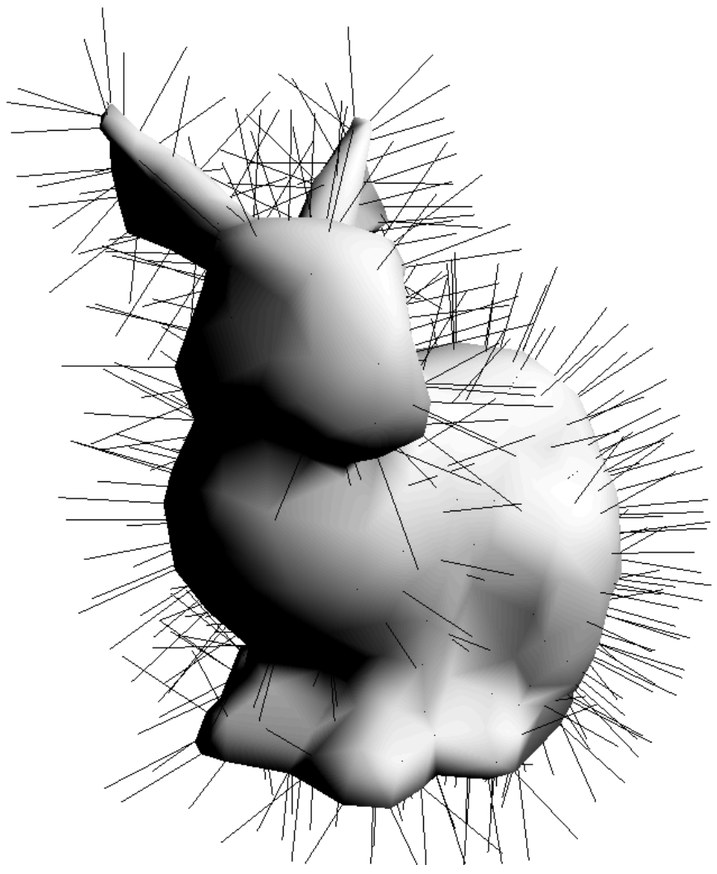
# Geometry shaders och Tesselation shaders

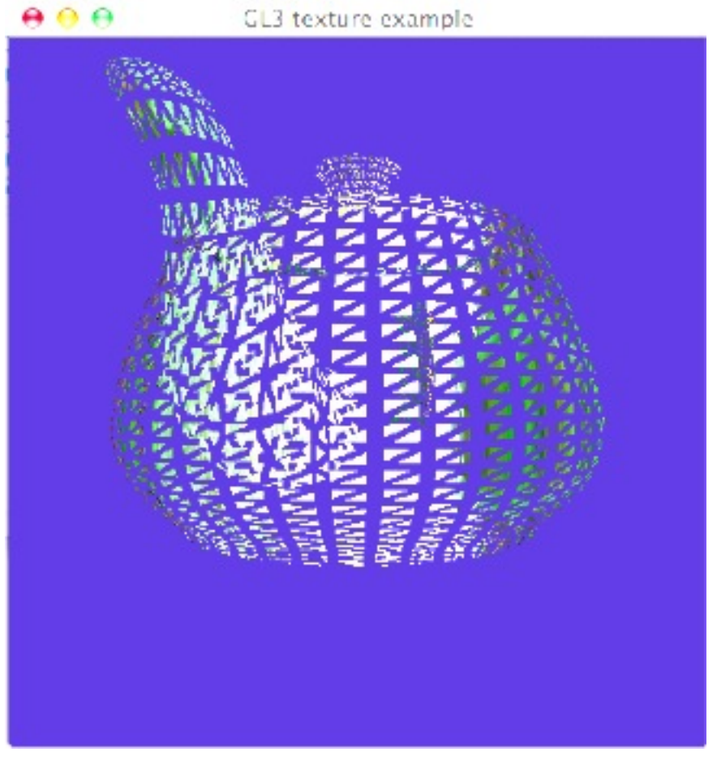
Additional shader stages in the geometry stage in the OpenGL pipeline

Can modify, add and remove geometry.

Can produce other kinds of geometry than was is entered.

# Geometry shaders





# Geometry shaders

OpenGL 3 (extension in GL 2)

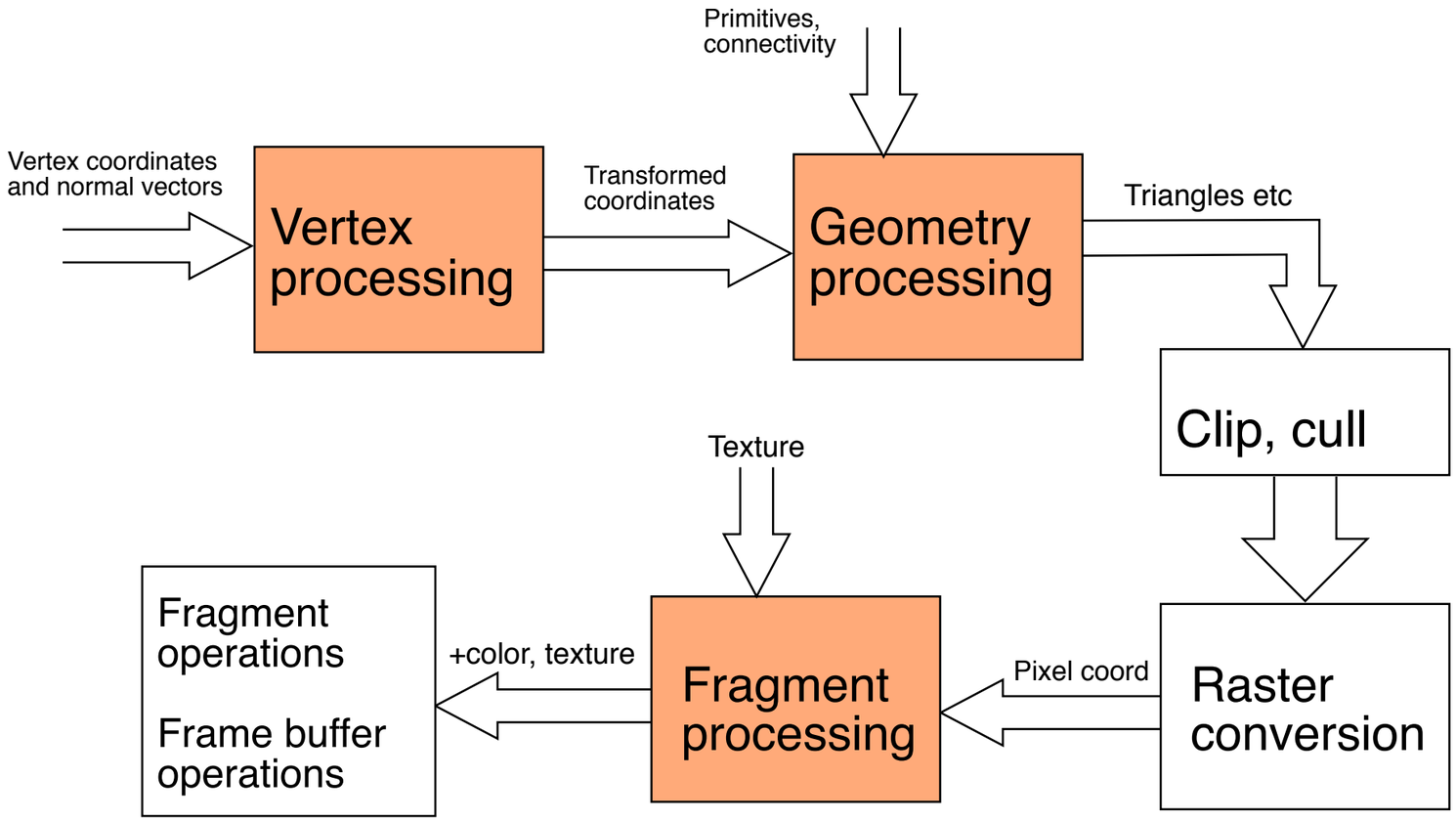
Shader between vertex and fragment, processes geometry, can add new geometry

Modest hardware demands: G80 or better (2007+)

Core since GL3.

Some limitations, had poor performance for some applications. Improved on newer GPUs!

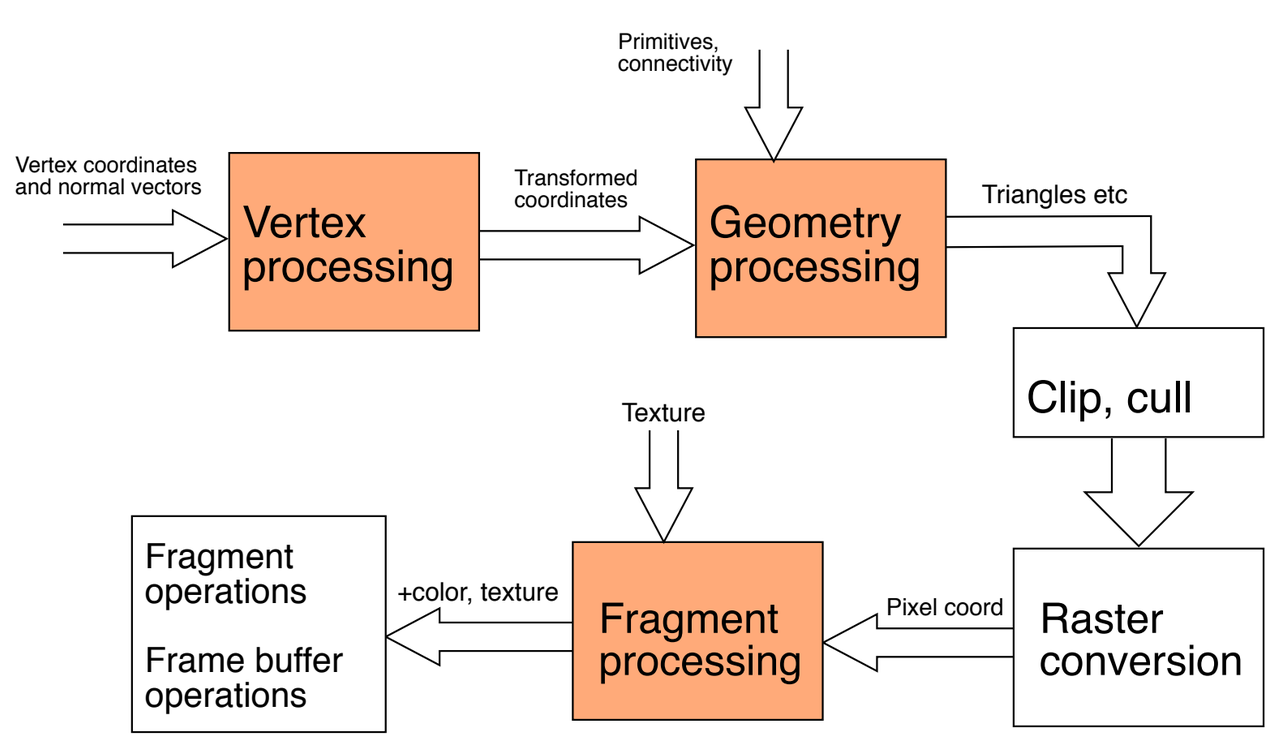
# OpenGL pipeline



# Geometry stage:

Shaders can modify, add or remove geometry.

Unlike the vertex stage it works on entire primitives.



# Applications for geometry shaders:

- Splines/spline-ytor
- Edge extraction, silhouettes
- Shadow volumes
- Effects on polygon level (e.g. breaking up a mesh by shrinking triangles)
- Dynamic hair/grass defined from a set of "root points".
- Adaptive subdivision (incuding displacement mapping)
- Visualization of normal vectors etc
- Flat shading
- Wireframes

# Geometry shader example

Input: A single triangle (as first example)

Load geometry shader together with vertex and fragment.

Indata to shader: triangles, lines or points

Output: triangle strips, line strips

# Pass-through geometry shader

```
#version 150

layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

void main()
{
  for(int i = 0; i < gl_in.length(); i++)
  {
    gl_Position = gl_in[i].gl_Position;
    EmitVertex();
  }
}
```

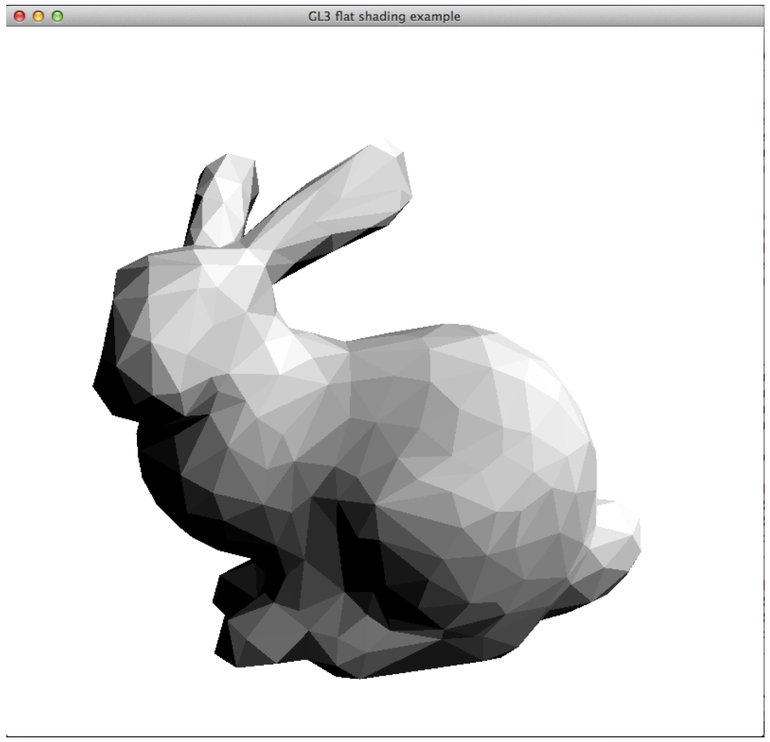
Pass on same vertex

Report that a primitive is finished!

# Flat shading

Easy with geometry shaders: Take the average of all vertex normals in GS, calculate light from that.

```
void main()
{
  vec4 middleOfTriangle = vec4(0.0);
  vec3 avgNormal = vec3(0.0);;

  for(int i = 0; i < gl_in.length(); i++)
  {
    avgNormal += exNormal[i];
  }
  middleOfTriangle /= gl_in.length();
  avgNormal /= gl_in.length();
  avgNormal = normalize(avgNormal);

  for(int i = 0; i < gl_in.length(); i++)
  {
    gl_Position = gl_in[i].gl_Position;
    texCoordG = texCoord[i];
    exNormalG = avgNormal;
    EmitVertex();
  }
  EndPrimitive();
}
```




A bit of a hassle to create something trivial...

# Wireframe

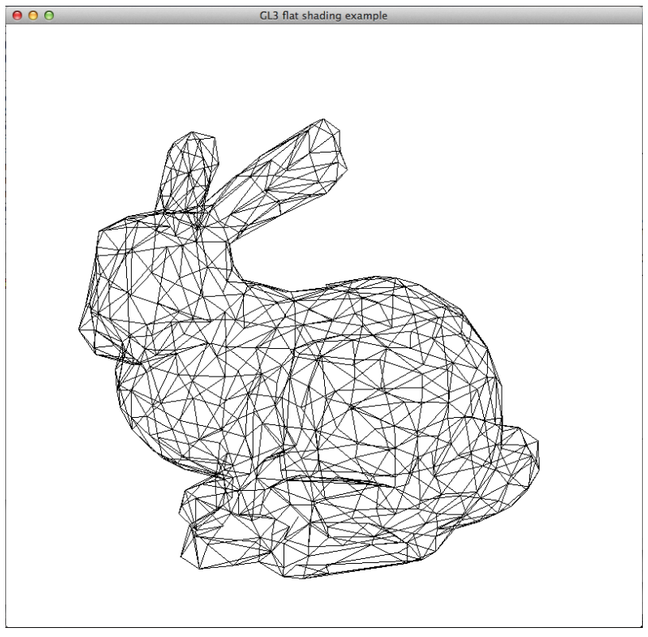
Wireframes can be generated by the geometry shader directly from polygons.

Convert every triangle to line strips in GS.

Very useful for visalizing geometry.

# Easy: line_strip out instead of triangle_strip!

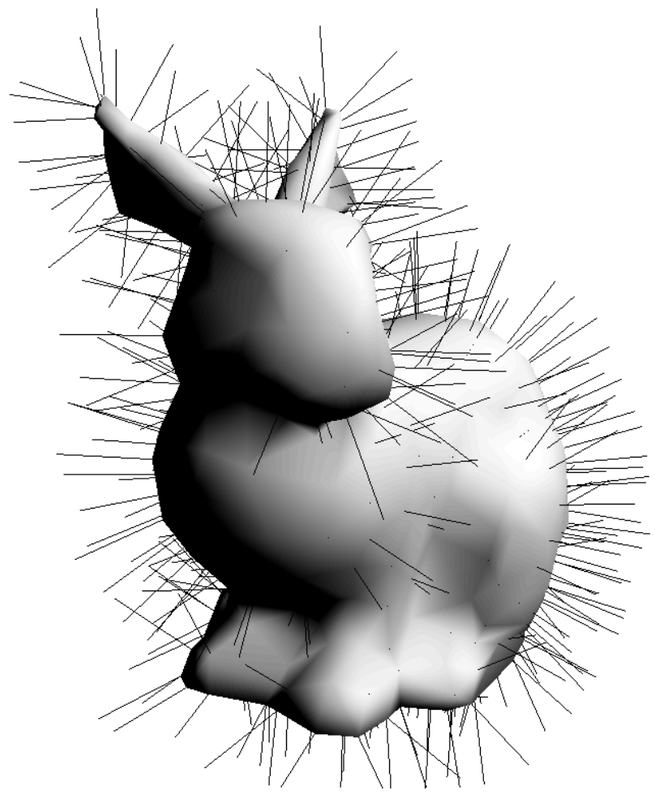
```
layout(triangles) in;
//layout(triangle_strip, max_vertices
    = 90) out; // Normal, solid
layout(line_strip, max_vertices = 90)
    out; // Wireframe
```



# Hair and grass

Use the geometry as staring points, generate lines/curves from these

```
for(float u = 0.0; u < 1.0; u += offs)
for(float v = 0.0; u+v < 1.0; v += offs)
{
   float u0 = u;
   float u1 = u + offs;
   float v0 = v;
   float v1 = v + offs;
   emit(u0,v0);
   EndPrimitive();
}
```



(The figuren only shows normal vectors)

# Crack shader

Not so useful but an amusing effect. Move all vertices closer to the center of the triangle!

```
void main()
{
  vec4 middleOfTriangle = vec4(0.0);
  float tw = 1.0 / offs;

  for(int i = 0; i < gl_in.length(); i++)
    middleOfTriangle += gl_in[i].gl_Position;
  middleOfTriangle /= gl_in.length();

  for(int i = 0; i < gl_in.length(); i++)
  {
    gl_Position = (gl_in[i].gl_Position * offs) +
(middleOfTriangle) * (1.0 - offs);

    texCoordG = texCoord[i];
    exNormalG = exNormal[i];
    EmitVertex();
  }
  EndPrimitive();
}
```

# How about tesselation...?

Add more geometry in order to

1) make a rough polygon model smoother

2) add detail (e.g. displacement mapping etc)

3) Handle level-of-detail

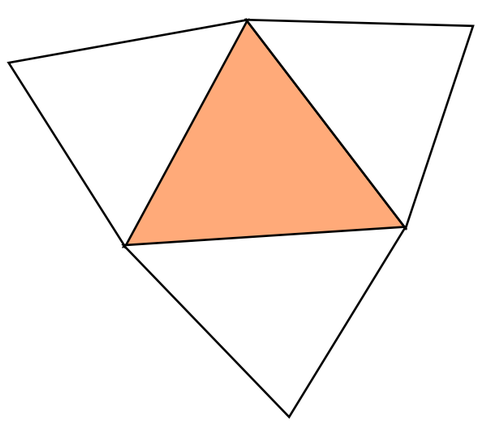
Often based on splined

Can be performed both in geometry and tesselation shaders

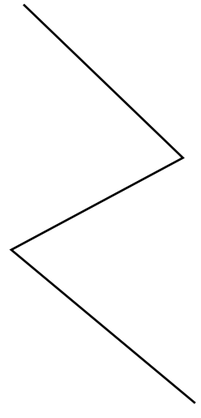
# Adjacency

Geometry can be passed together with information of neighbor geometry

This can be used e.g. for tesselation



Triangle with adjacency



Line with adjacency

More applications: Shadow volumes, edge extraction

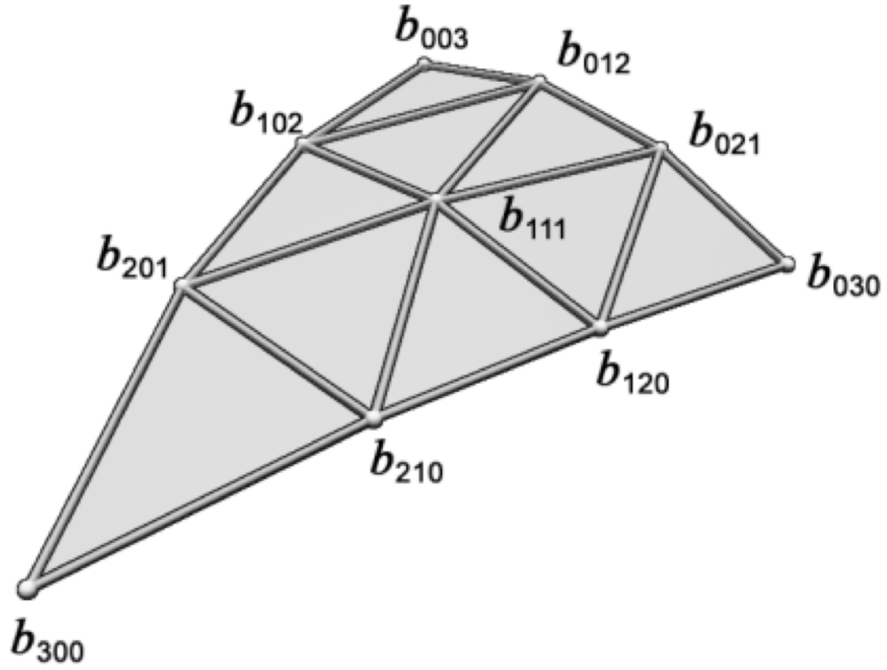
# Curved PN Triangles

Suitable for geometry shaders

Only needs vertices and normal vectors for one triangle at a time. Not even adjacency! Comfortable.

PN = Point + Normal

# Curved PN Triangles



# Instancing av geometry shaders

Performance problem when few elements go in and many out!

Instancing run g.s. several times per input primitive!

layout(invocations = num_instances) in;

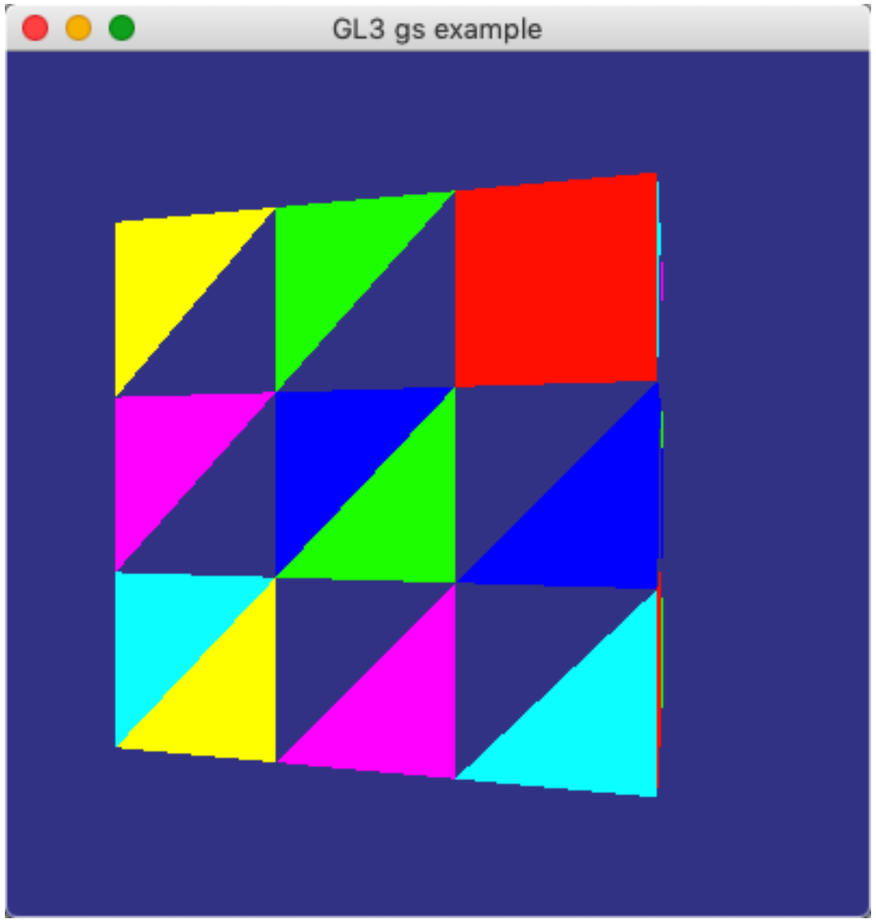
gl_InvocationID

At least 32 instances are guaranteed to be possible!

Standard from OpenGL 4.0!

# Geometry shader instancing example

```
#version 410

layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;
layout(invocations = 6) in;

out vec4 exColor;

void main()
{
  vec4 middleOfTriangle = vec4(0.0);
  vec4 v1,v2,v3;

  // Every triangle is split in 6 triangles
  // but only one is made by each instance.
  // Pretty stupid; no claims of good performance.

  switch (gl_InvocationID)
  {
     case 0:
       v1 = gl_in[0].gl_Position;
       v2 = (gl_in[0].gl_Position*2 + gl_in[1].gl_Position)/3;
       v3 = (gl_in[0].gl_Position*2 + gl_in[2].gl_Position)/3;
       exColor = vec4(1,0,0,1);
       break;
     case 1:
```


GL3 gs example

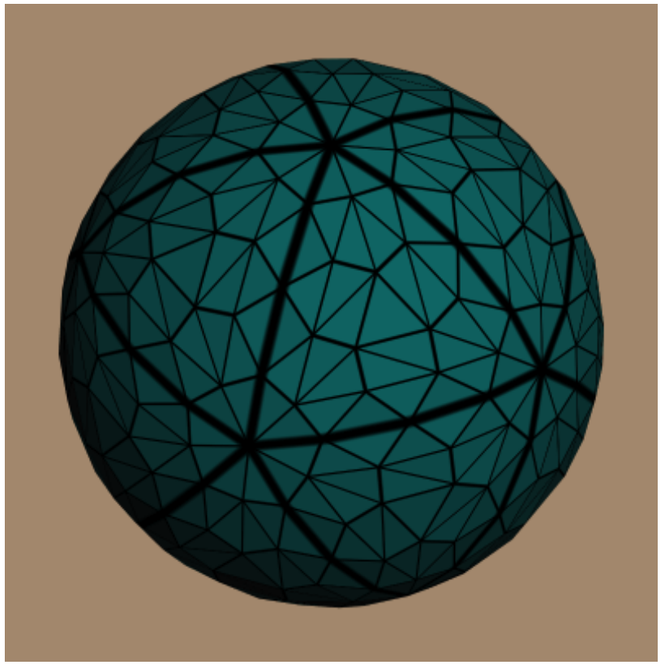
# Conclusions

Geometry shaders are highly useful with many applications.

One extra shader stage does not complicate things much.

Enough for a lot - but not everything.

# Tesselation shaders



# Tesselation shaders

Problem with early geometry shaders:
Much data from little indata inefficient, serialized.

Remedy 1: Instancing in newer hardware, faster geometry shaders

Remedy 2: Tesselation shaders

# Tesselation shaders

Located between the vertex shader and the geometry shader

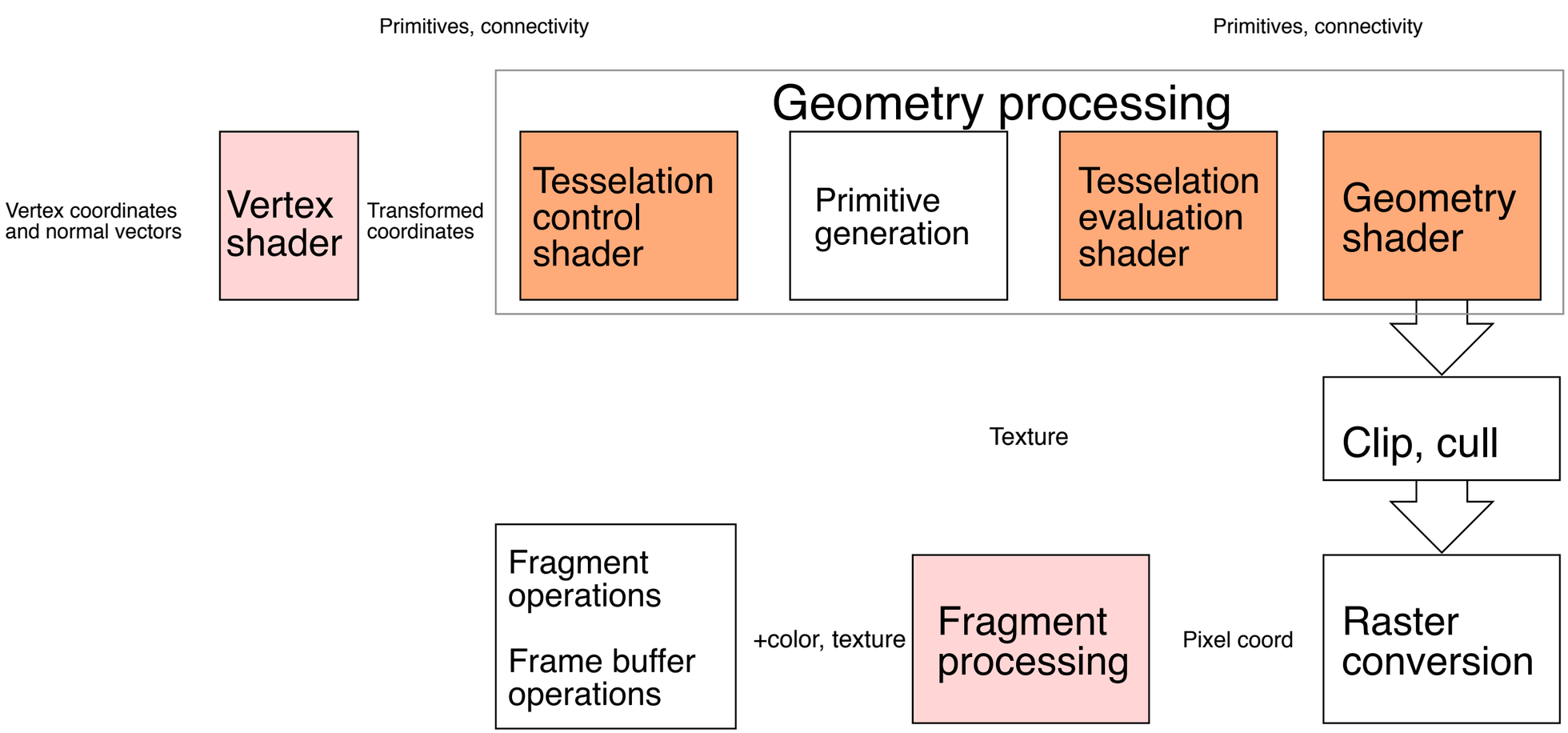
Not one shader but two, plus non-programmable stages between.

1) Tesselation control shader

2) Primitive generation

3) Tesselation evaluation shader

# OpenGL pipeline - extended



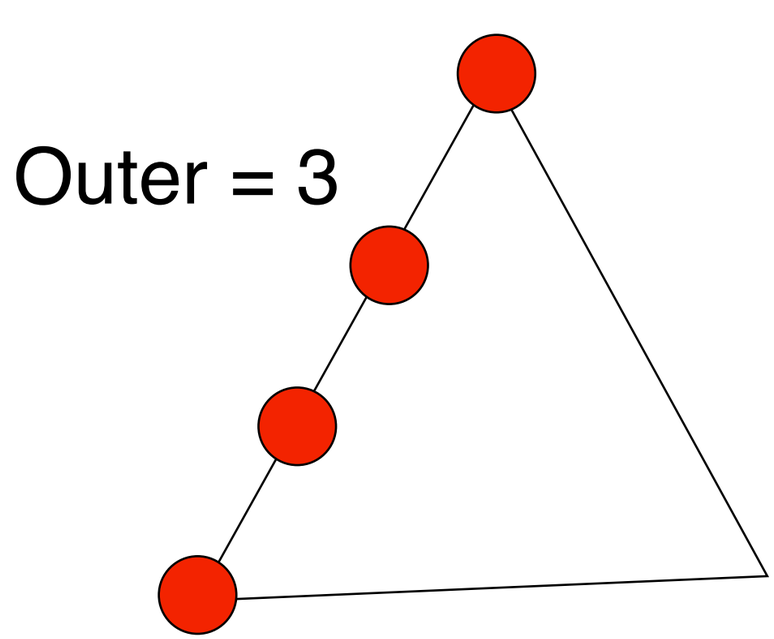
# Tasks for tesselation shaders

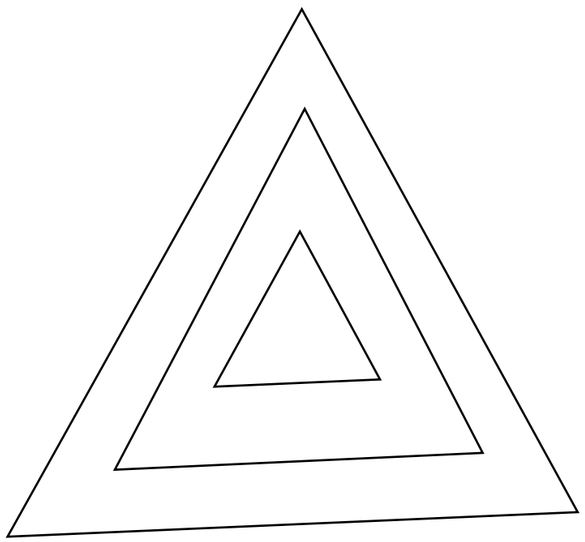
Indata: ”Patches”, a number of vertices without any given constellation.

The TC shader sets a desired tesselation level per edge and number of inner levels.

The TE shadern calculated the final positions in the desired way (e.g. using a spline)

# Example: Triangel



Inner = 3

# Example: Tesselation Control

```
#version 400
layout(vertices = 3) out;
in vec3 vPosition[]; // From vertex shader
out vec3 tcPosition[]; // Output of TC

uniform float TessLevelInner; // Sent from main program
uniform float TessLevelOuter;

void main()
{
    tcPosition[gl_InvocationID] = vPosition[gl_InvocationID]; // Pass on vertex
    if (gl_InvocationID == 0)
    {
        gl_TessLevelInner[0] = TessLevelInner; // Decide tesselation level
        gl_TessLevelOuter[0] = TessLevelOuter;
        gl_TessLevelOuter[1] = TessLevelOuter;
        gl_TessLevelOuter[2] = TessLevelOuter;
    }
}
```

# Example: Tesselation Evaluation

```
#version 400

layout(triangles, equal_spacing, cw) in;
in vec3 tcPosition[]; // Original patch vertices

void main()
{
    vec3 p0 = gl_TessCoord.x * tcPosition[0]; // Barycentric!
    vec3 p1 = gl_TessCoord.y * tcPosition[1];
    vec3 p2 = gl_TessCoord.z * tcPosition[2];
    gl_Position = vec4(p0 + p1 + p2, 1);
// Sum with weights from the barycentric coords any way we like

// Apply vertex transformation here if we want
}
```

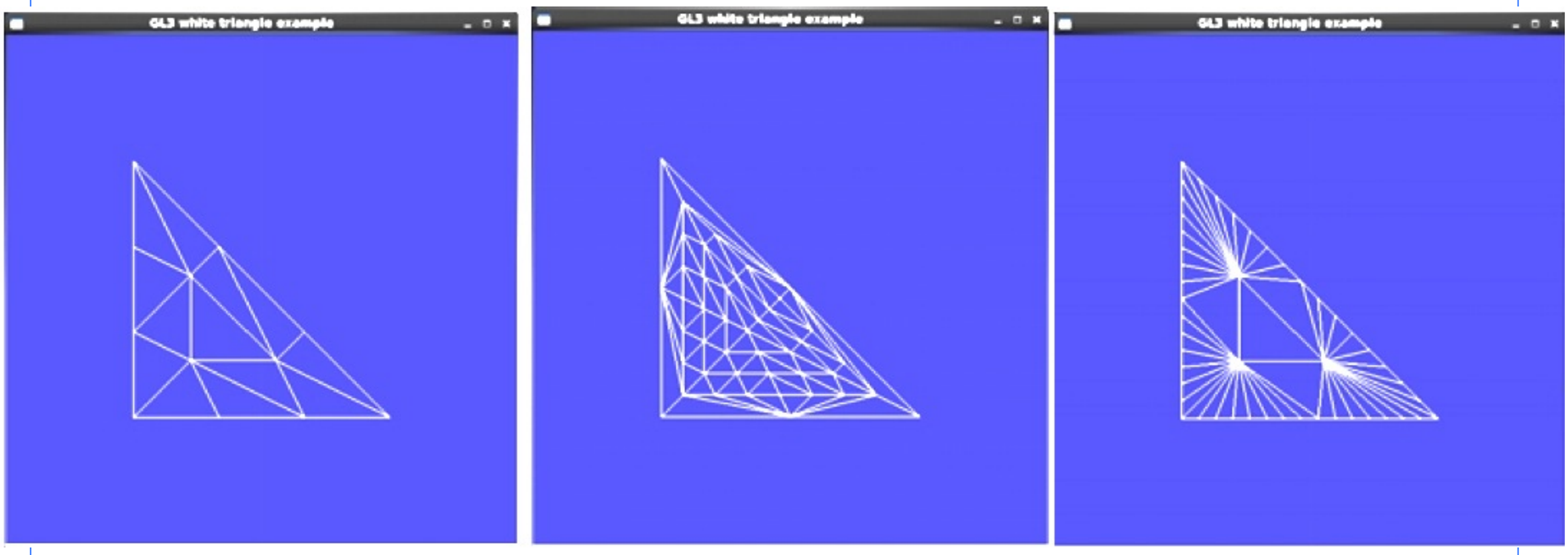
# Control + Evaluation

Control decides how many levels of tesselation that is desired.

Evaluation is called multiple times. Each time we get unique coordinated from which we should calculate the resulting vertex position

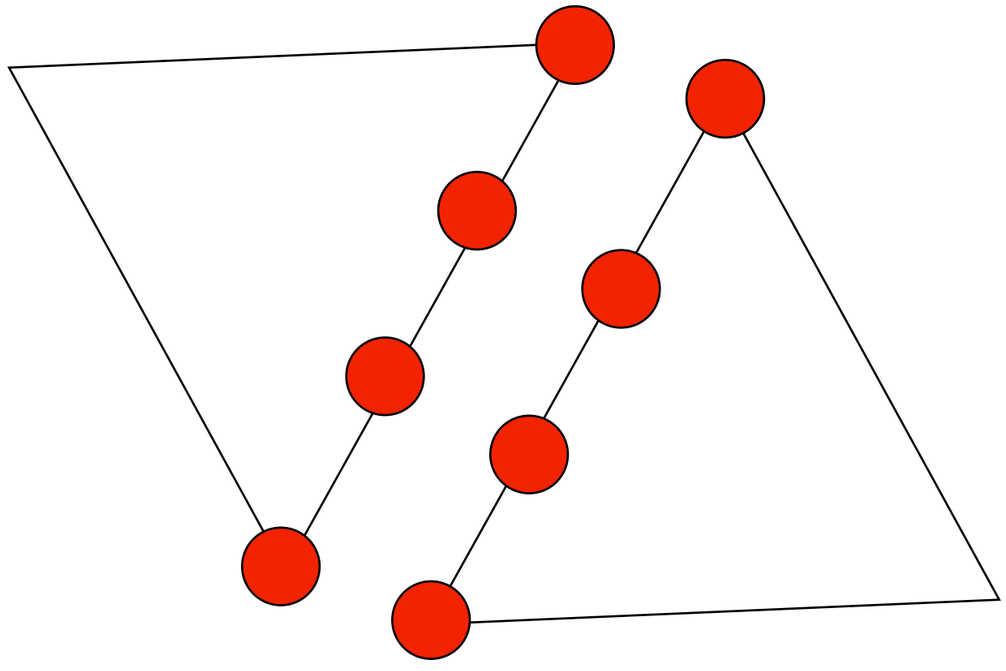
# Result

# Variation by outer and inner



# ”Outer” is by edge

## Important!



If you calculate the edge in a way that gives the same resolution in both patches for an edge shared by neighbor patches, you can avoid gaps!

# Old style tesselation, "Evaluators"

Old OpenGL had ”Evaluators” (glMap)

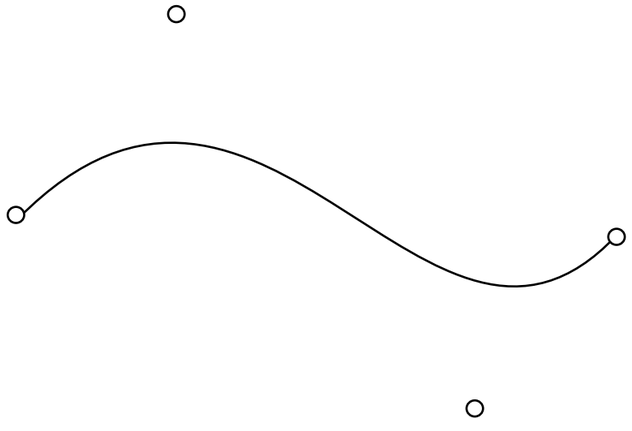
Good support for Bezier curves and Bezier surfaces.

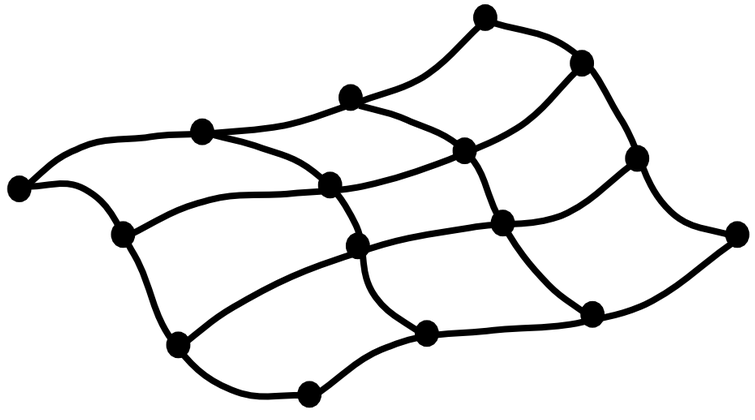
Less control than the modern tesselation.

# Evaluators

Old API for evaluating Bezier curves. Easy to use but now deprecated.

Both curves and surfaces.





# Evaluators

Rättframt men inte så flexibelt

```
glMap1f(GL_MAP1_VERTEX_3, u0, u1, 3, 4, &data2[0][0]);
glEnable(GL_MAP1_VERTEX_3);
glBegin(GL_LINE_STRIP);
for (int i = 0; i <= 20; i++)
glEvalCoord1f(u0 + i*(u1-u0)/20);
glEnd();
```

Control points

Evaluation, specifies vertices

# Evaluators

Old-fashioned! Only quads, tricky to use on general models.

=> go for geometry shaders and tesselation shaders!

# Conclusions on tesselation shaders

Double stages - mode complicated than geometry shaders

Vertex shader gets superfluous

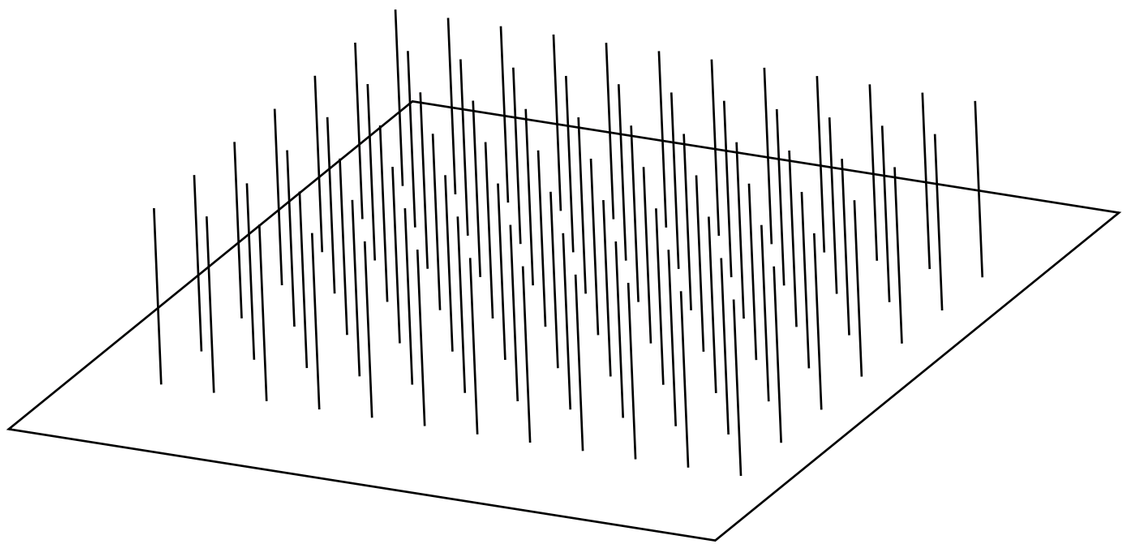
The point is tesselation, not much else. More applications for Geometry shaders?

Do we have enough shader stages yet?

# Grass and fur

Nice application of geometry shaders

Spawn strands over given polygons

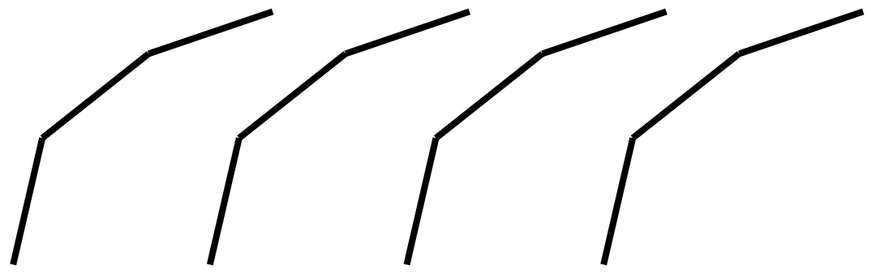


# Bending grass

Bend, wave with wind - use several sections

Simple case or *tropism*

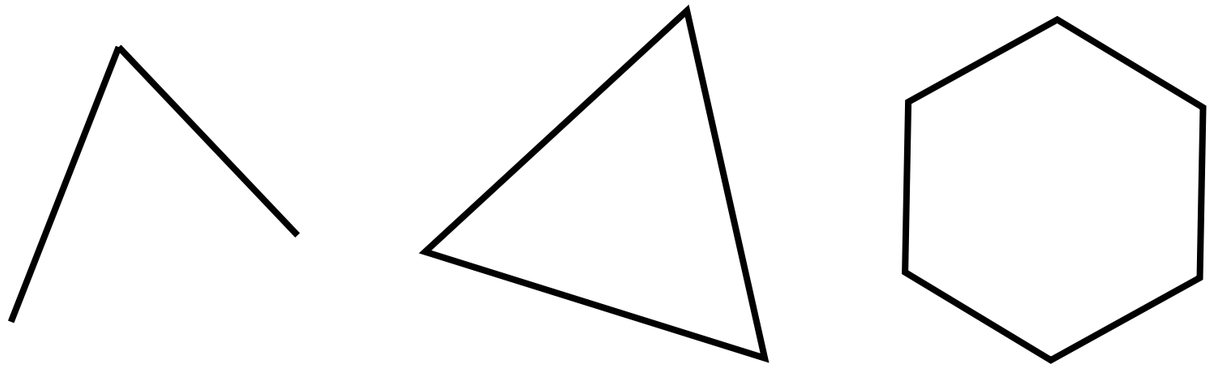
The wind needs to vary (consider harmnic functions) to vary the bend over the grass.



# Don't make it disappear

Flat polygons get invisible when watches from the side

It needs thickness



Just two edges (4 triangles per section)

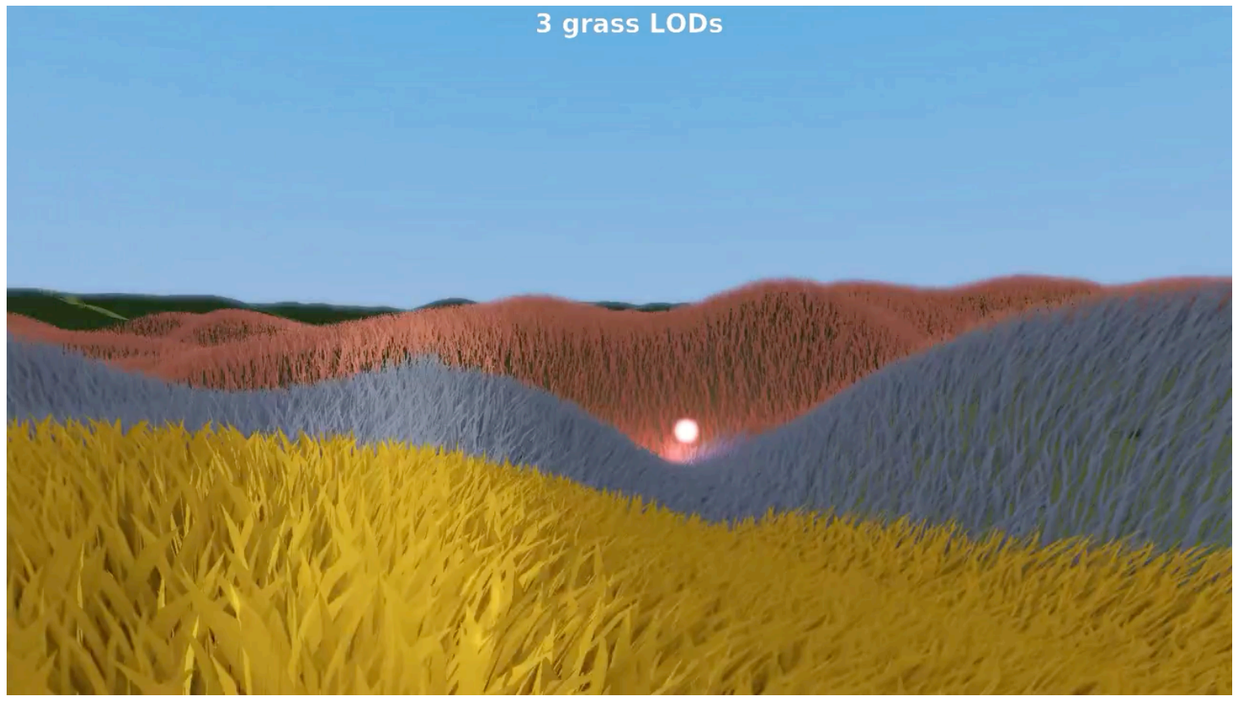
Triangular pipe

More detailed pipe

# Level of detail on distance

Fewer strands

or switch to billboards (more about these next time)



# Light

Make the grass darker by depth to approximate the occlusion effect

You can also use ambient occlusion (dampen based on proximity)

# More realistic trees

Tropism

Organic growth, dependency of environment

# Organic growth

How does the tree grow?

- Amount of light
- Amount of water
- Physical obstructions

# Trees do not always change much due to external force

"Övervallning"
in swedish

# Tropism

Adjust the direction of branches due to external forces.

Main force: Gravity. Other possibilities, like wind.

Gravity is constant -> Can be included in pre-generation.

For every branch, inspect the local orientation (given by the current matrix). Change the bend in the right direction.

Also relevant for grass, though mainly for wind.

# Branch direction

The "forward vector" of a branch is given by its rotation matrix!

You can lift out the vector directly from the matrix!

M * (0, 1, 0) = vector along current Y

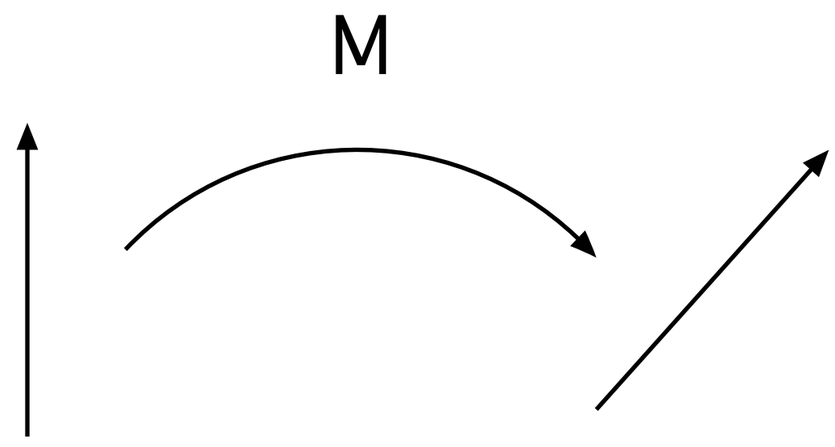
Given that vector, rotate a suitable amount to adjust for gravity!

# Rotation to rotation

Model coordinates, can be the Y direction

Branch direction = M * (0,1,0)

So M holds all information you need! It tells you how much rotation it has. Use that to find a modification for tropism. You want *more* rotation. How much?



# Lab 4: Planet generation

Tesselation lab!

Start from a cube.

Tesselate.

Change shape to a sphere.

Add noise for detail.

Take a walk on your planet.

# Trying to make it managable:

You start with all shaders in place!

No guessing on basic syntax, it is all there.

Modify to solve tasks.

but as always, there will be unexpected things in a new lab...