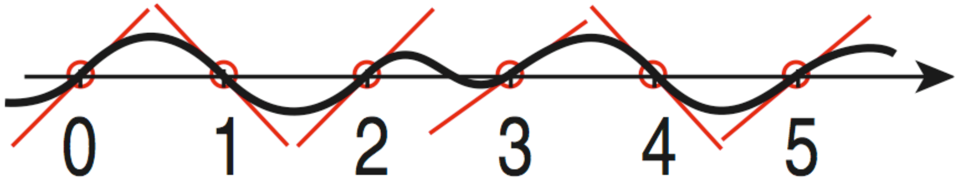
Better Noise

- *Gradient noise*:
  - Specify the gradients at integer points (instead of at values):

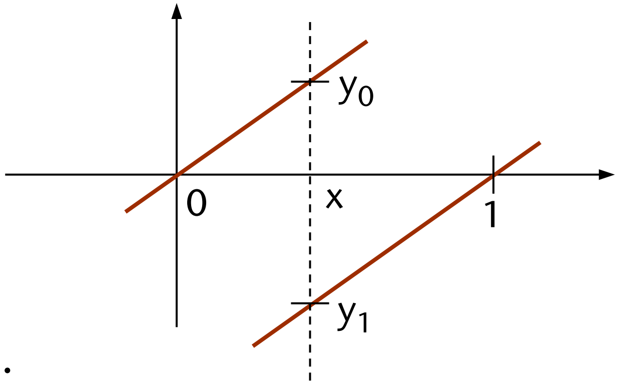


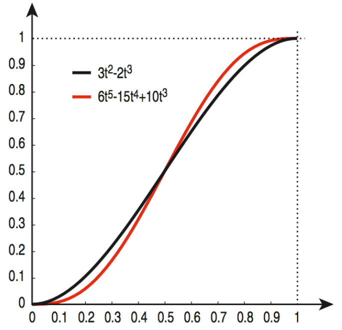
  - Interpolation:
    - At position x, calculate y_0 and y_1 as values of the lines through x=0 and x=1 with the previously spec'd (random) gradients
    - Interpolate y_0 and y_1 with a blending function, e.g.

$$h(x) = 3x^2 - 2x^3$$

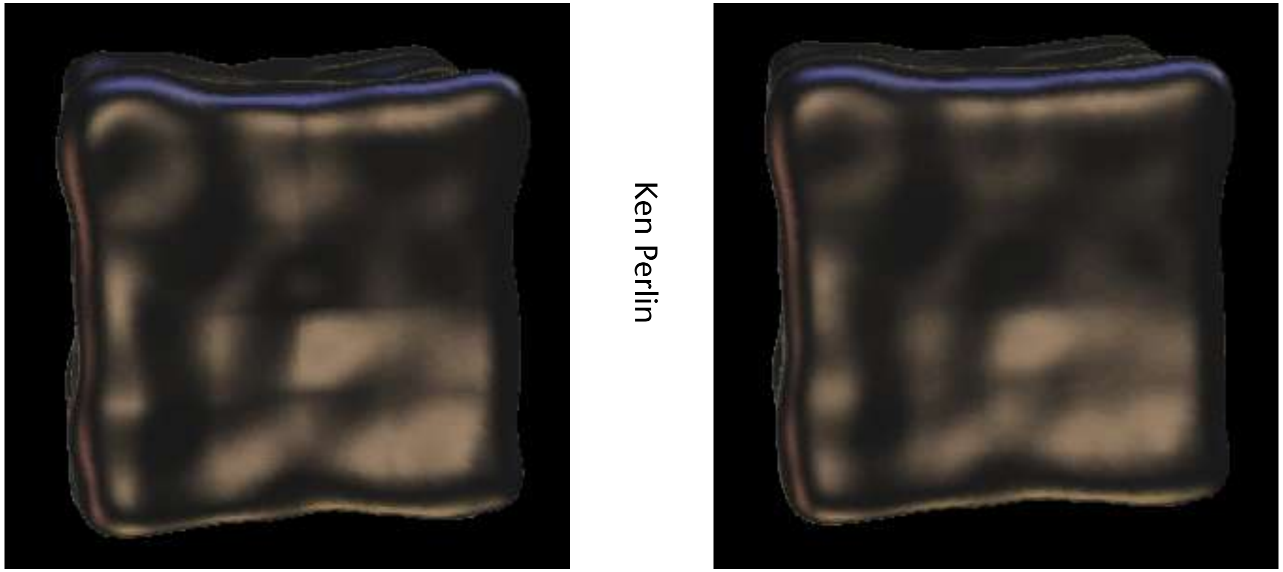
or

$$q(x) = 6x^5 - 15x^4 + 10x^3$$





- Advantage of the quintic blending function: second derivative at x=0 and x=1 is 0 → the entire noise function is C^2-continuous
- Example where one can easily see this:



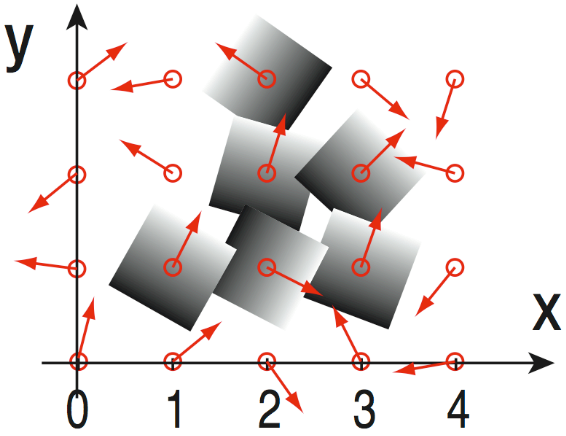
Ken Perlin

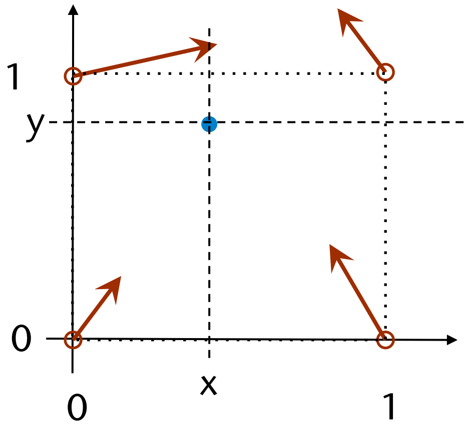
Cubic interpolation   Quintic interpolation

- Gradient noise in 2D:
  - Set gradients at *integer grid points*
    - *Gradient* = 2D vector, not necessarily with length 1
  - Interpolation (as in 1D):
    - W.l.o.g., $P = (x,y) \in [0,1]\text{x}[0,1]$
    - Let the following be the gradients:
      g_{00} = gradient at (0,0), g_{01} = gradient at (0,1),
      g_{10} = gradient at (1,0), g_{11} = gradient at (1,1)
    - Calculate the values z_{ij} of the "gradient ramps" g_{ij} at point P :

$$z_{00} = g_{00} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \qquad z_{10} = g_{10} \cdot \begin{pmatrix} x-1 \\ y \end{pmatrix}$$

$$z_{01} = g_{01} \cdot \begin{pmatrix} x \\ y-1 \end{pmatrix} \qquad z_{11} = g_{11} \cdot \begin{pmatrix} x-1 \\ y-1 \end{pmatrix}$$



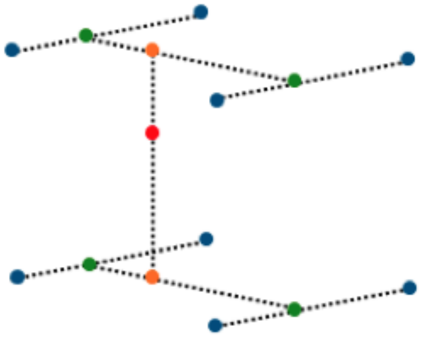


- Blending of 4 z-values through bilinear interpolation:

$$z_{x0} = (1 - q(x))z_{00} + q(x)z_{10} \;, \qquad z_{x1} = (1 - q(x))z_{01} + q(x)z_{11}$$

$$z_{xy} = (1 - q(y))z_{x0} + q(y)z_{x1}$$

- Analogous in 3D:
  - Specify gradients on a 3D grid
  - Evaluate $2^3 = 8$ gradient ramps
  - Interpolate these with tri-linear interpolation and the blending function



Total LERPs = 4 + 2 + 1

= 7

- And in *d*-dim. space? → complexity is $O(2^d)$!

Simplex Noise

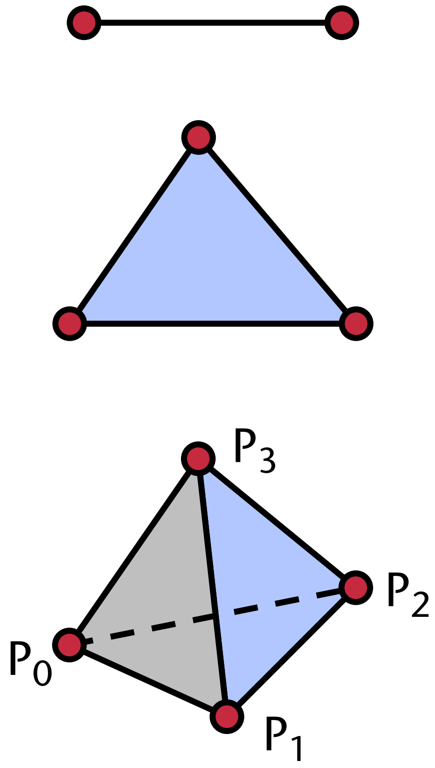
- d-dimensionaler simplex: =

  combination of $d+1$ affinely independent points
- Examples:
  - 1D simplex = line, 2D simplex = triangle,
    3D simpex = tetrahedron
- In general:
  - Points $P_0, \ldots, P_d$ are given
  - *d*-dim. simplex = all points *X* with

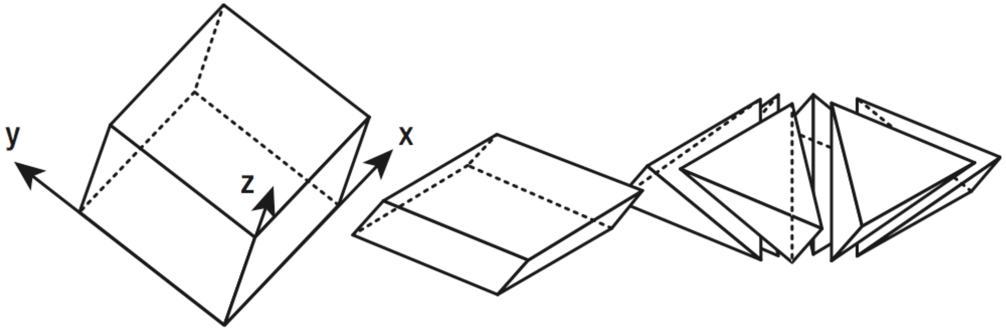
$$X = P_0 + \sum_{i=1}^{d} s_i \mathbf{u}_i$$

with

$$\mathbf{u}_i = P_i - P_0 \ , \ s_i \geq 0 \ , \ \sum_{i=0}^{d} s_i \leq 1$$

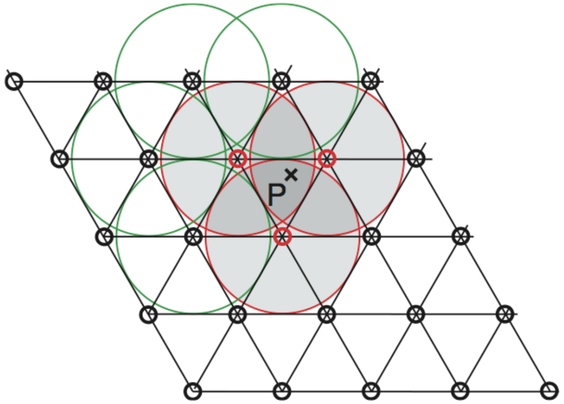


- In general, the following is true:
  - A *d*-dimensional simplex has *d*+1 vertices
  - With equilateral *d*-dimensional simplices, one can partition a cube that was suitably "compressed" along its diagonals

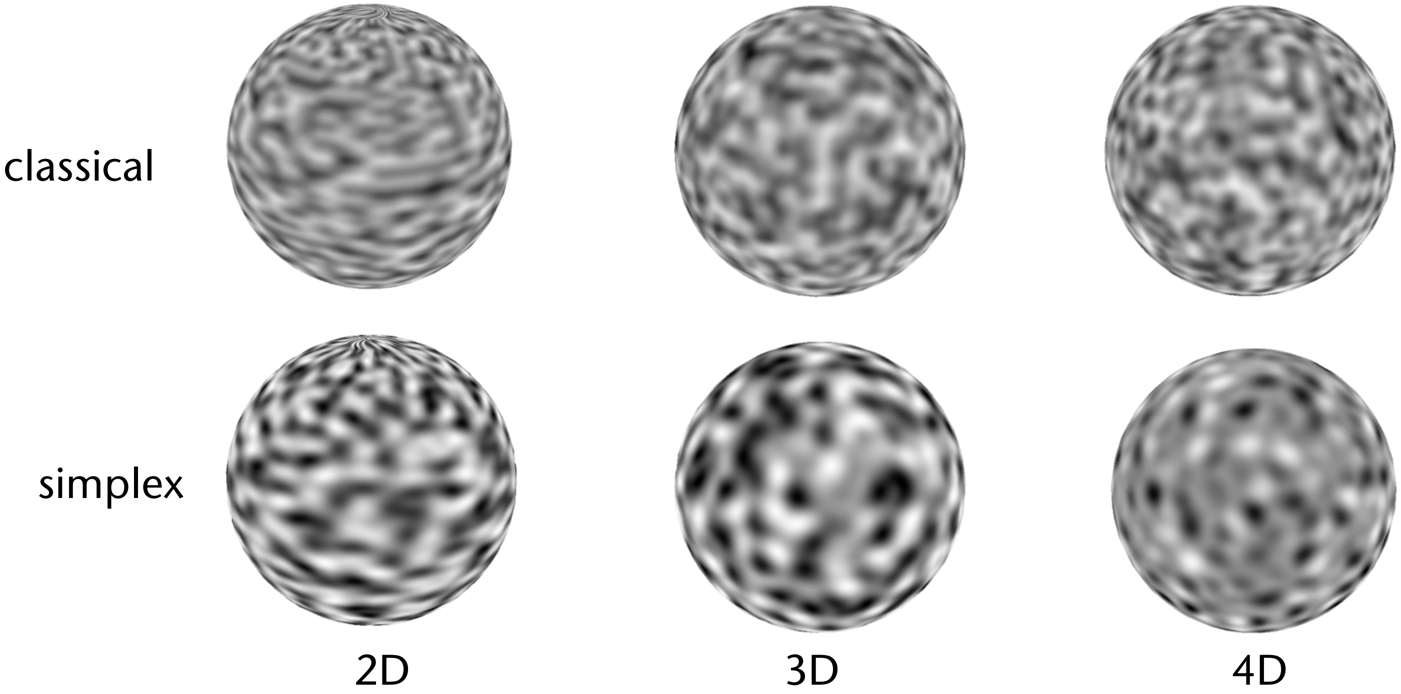


    - Such a "compressed" *d*-dimensional cube contains *d*! many simplices
  - Consequence: with equilateral *d*-dimensional simplexes, one can partition *d*-dimensional space (*tessellation*)

- Construction of the noise function over a simplex tessellation (hence *"simplex noise"*):
  - Determine the simplex in which a point *P* lies
  - Determine all of its corners and the gradients in the corners
  - Determine (as before) the value of these "gradient ramps" in *P*
  - Generate a weighted sum of these values
  - Choose weighting functions so that the “influence” of a simplex grid point only extends to the incidental simplexes



- A huge pro: has only complexity $O(d)$
- For details see "Simplex noise demystified" (on the course's homepage)
- Comparison between classical value noise and simplex noise:



- 4 noise functions are defined in the GLSL standard:

  `float noise1(gentype), vec2 noise2(gentype),`
  `vec3  noise3(gentype), vec4 noise4(gentype).`

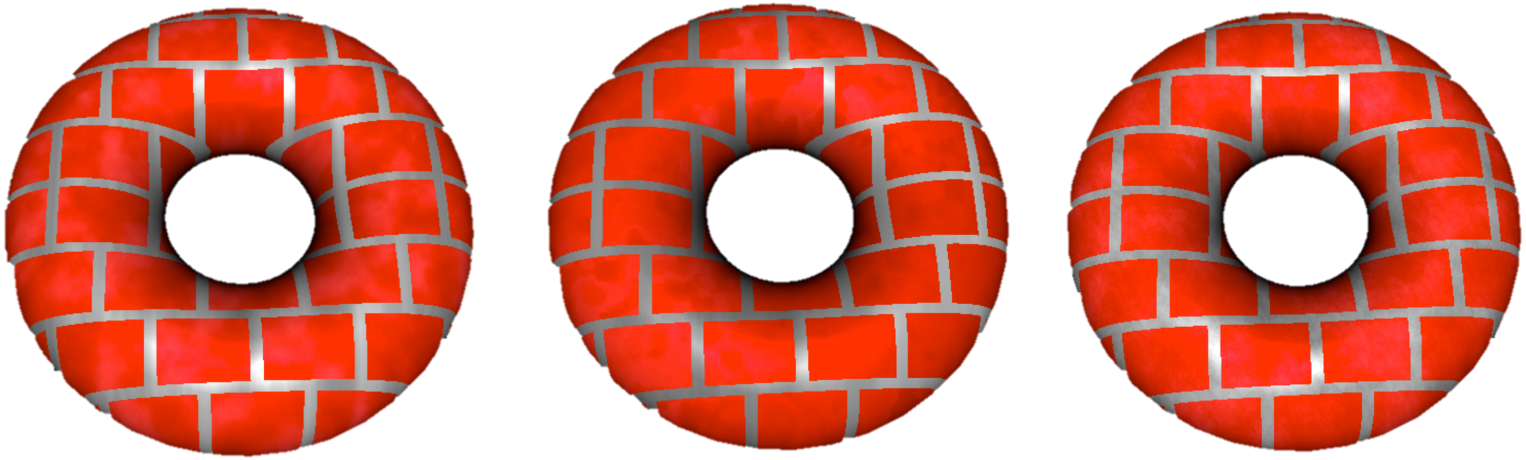
- Calling such a noise function:

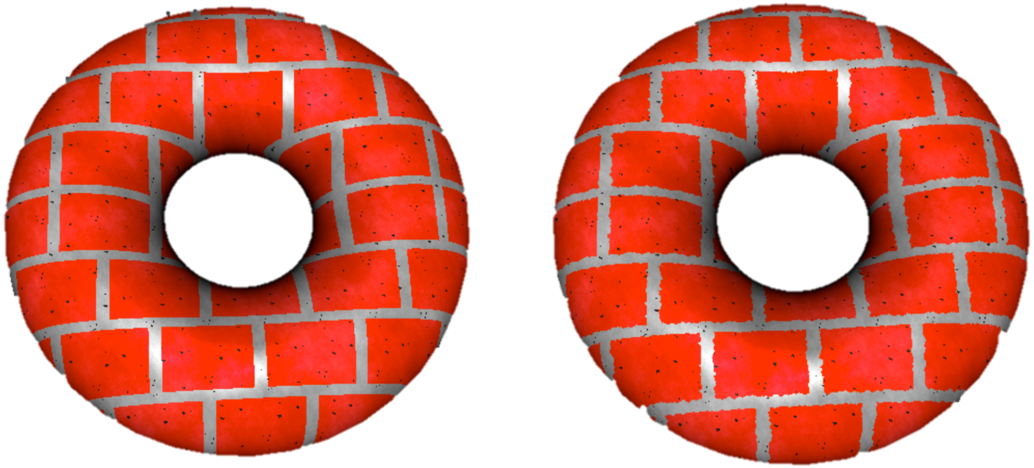
  $$v = \text{noise2}(\, f{*}x + t,\ f{*}y + t\,)$$

  - With *f*, one can control the spatial frequency,
    With *t*, one can generate an animation (*t*="time").
  - Analogous for 1D and 3D noise
- Caution: range is [-1,+1]!
- Cons:
  - Are not implemented everywhere
  - Are slooooooow…

Example: Application of Noise to our Procedural Texture

- Our procedural brick texture (please ignore the uneven outer torus contour, that's an artifact from Powerpoint):

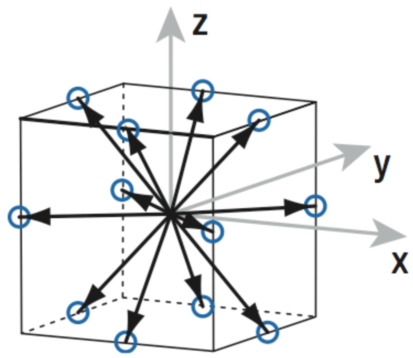




The code for this example is on the class's homepage: vorlesung_demos/ brick.vert and brick[4-8].frag

Remarks

- Goal: repeatable noise function
  - That is, f(x) always returns the same value for the same x
- Choose fixed gradients at the grid points
- Observation: a few different ones are sufficient
  - E.g. for 3D, gradients from this set are sufficient:

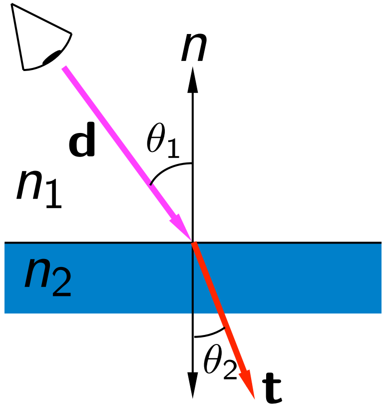


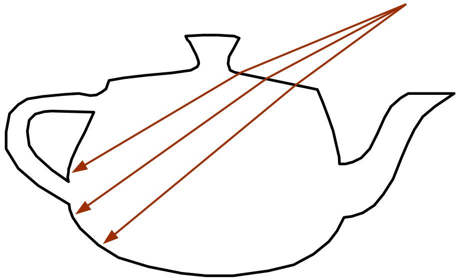
$$
\begin{aligned}
g_0 &= (0, 1, 1),\ g_1 = (0, 1, -1),\\
g_2 &= (0, -1, 1),\ g_3 = (0, -1, -1),\\
g_4 &= (1, 0, 1),\ g_5 = (1, 0, -1),\\
g_6 &= (-1, 0, 1),\ g_7 = (-1, 0, -1),\\
g_8 &= (1, 1, 0),\ g_9 = (1, -1, 0),\\
g_{10} &= (-1, 1, 0),\ g_{11} = (-1, -1, 0)
\end{aligned}
$$

- Integer coordinates of the grid points can be simply hashed→ index into a table of pre-defined gradients

Light Refraction

- With shaders, one can try approximations of simple global effects
- Example: light refraction
- What does one need to calculate the refracted ray?
  - Snell's Law: $n_1 \sin\theta_1 = n_2 \sin\theta_2$
  - Needed: **n**, **d**, n_1, n_2
  - Everything is available in the fragment shader
  - So, one can calculate **t** *per pixel*
- So why is refraction so difficult?
  - In order to calculate the correct cutting point of the refracted ray, one needs the entire geometry!

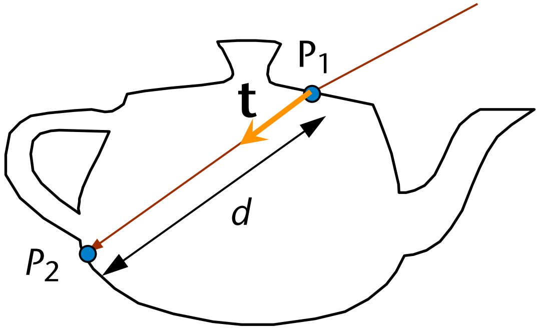


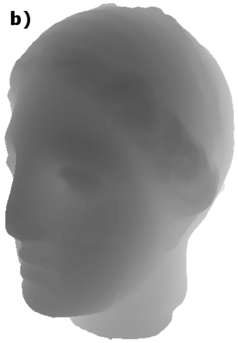


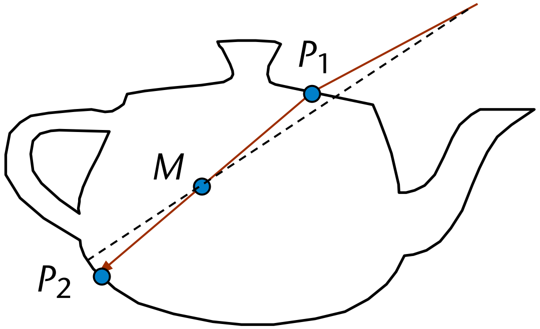
- Goal: approximate transparent object with two planes, which the incoming & refracted rays intersect
- Step 1: determine the next intersection point

$$P_2 = P_1 + d\mathbf{t}$$

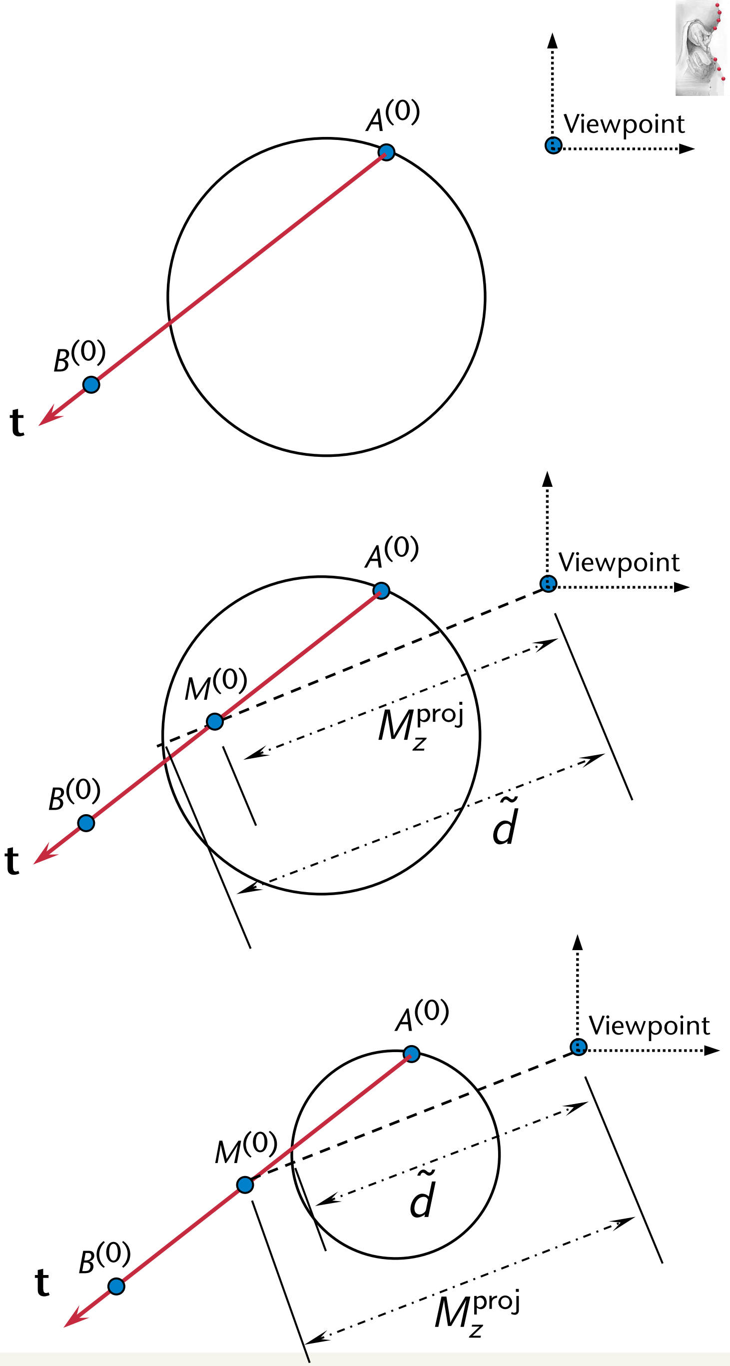
  - Idea: approximate *d*
  - To do that, render a depth map of the back-facing polygons in a previous pass, from the viewpoint
  - Use binary search to find a good approximation of the depth (ca. 5 iter.)



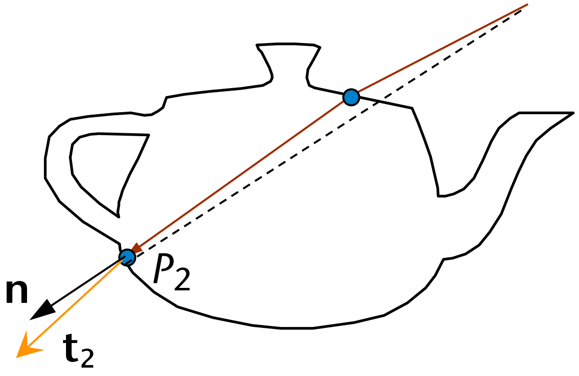


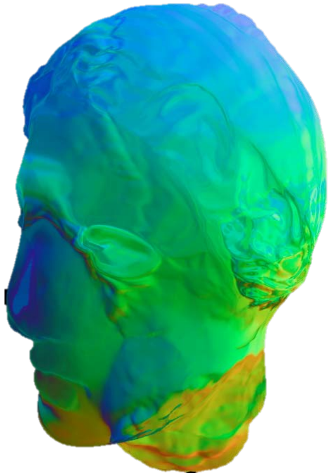


- On the binary search for finding the depth between P_1 and P_2:
  - Situation: given a ray **t**, with $t_z < 0$, and two "bracket" points $A^{(0)}$ and $B^{(0)}$, between which the intersection point must be; and a precomputed depth map
  - Compute midpoint $M^{(0)}$
  - Project midpoint with projection matrix $\longrightarrow M^{\text{proj}}$
  - Use $(M_x^{\text{proj}}, M_y^{\text{proj}})$ to index the depth map $\longrightarrow \tilde{d}$
  - If $\tilde{d} > M_z^{\text{proj}} \Rightarrow$ set $A^{(1)} = M^{(0)}$
  - If $\tilde{d} < M_z^{\text{proj}} \Rightarrow$ set $B^{(1)} = M^{(0)}$



- Step 2: determine the normal in P_2
  - To do that, render a normal map of all back-facing polygons from the viewpoint
  - Project P_2 with respect to the viewpoint into screen space
  - Index the normal map
- Step 3:
  - Determine $\mathbf{t}_2$
  - Index an environment map

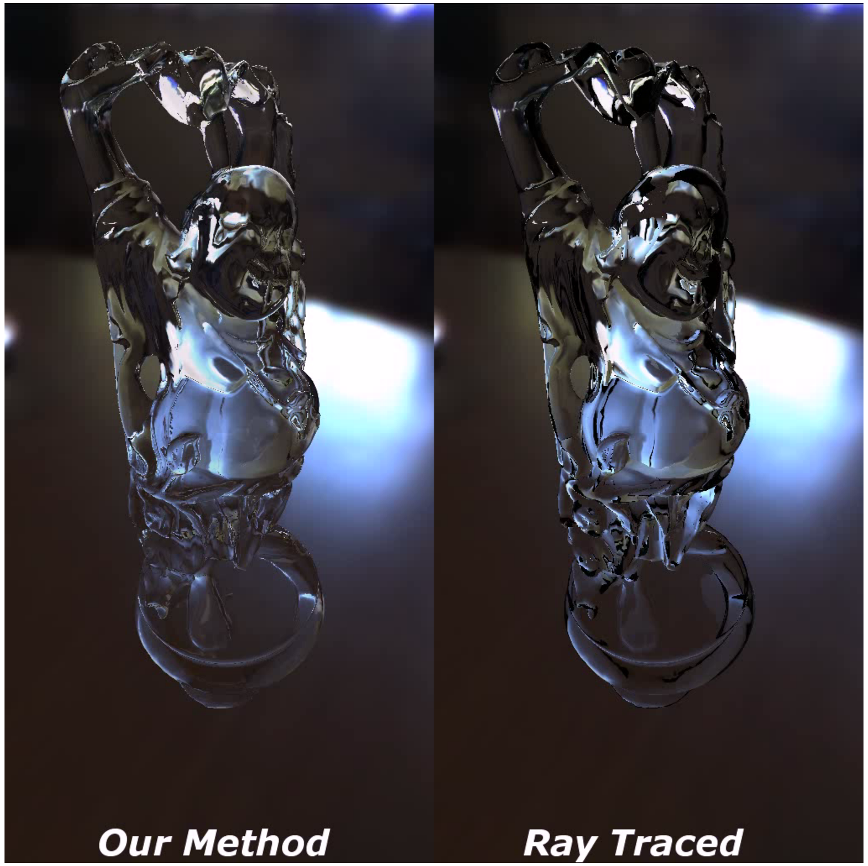




Normal map

- Many open challenges:
  - When *depth complexity* > 2:
    - Which normal/which depth value should be stored in the depth/normal map?
  - Approximation of distance
  - Aliasing

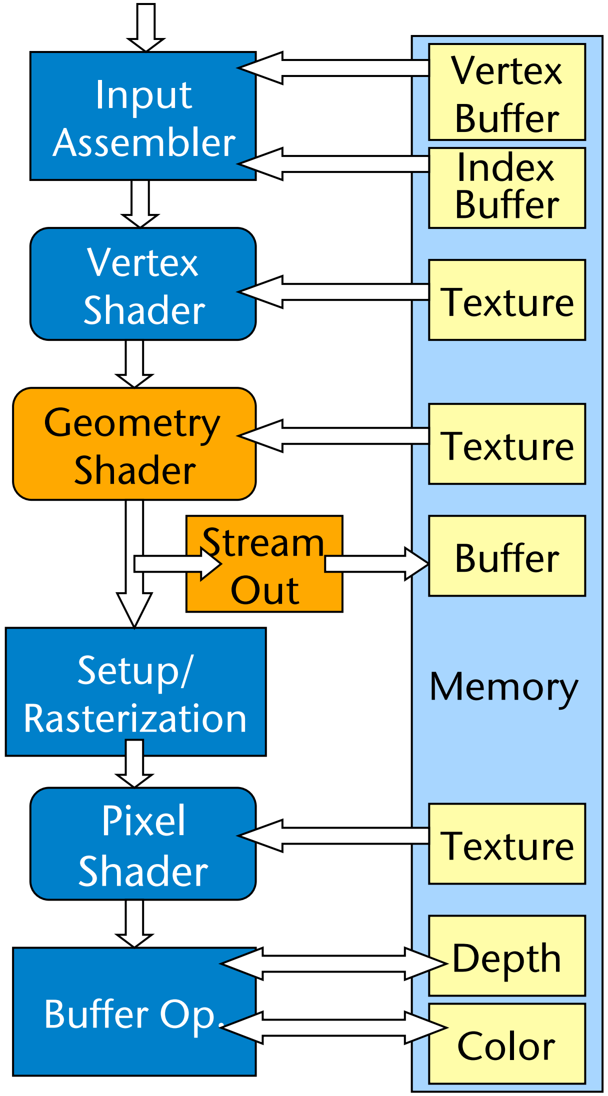
Examples

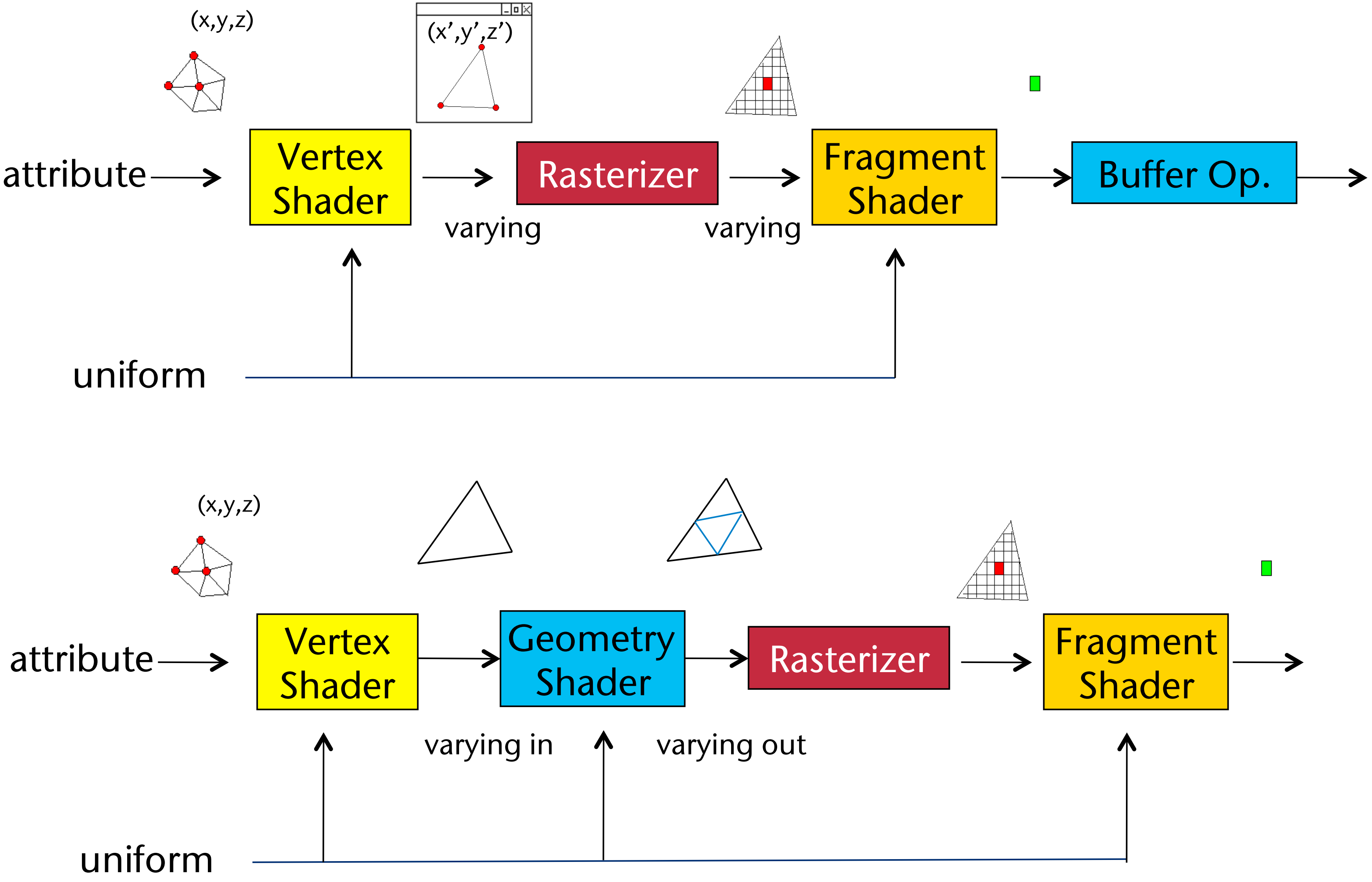


With internal reflection

The Geometry Shader

- Situated between vertex shader and rasterizer
- Essential difference to other shaders:
  - Per-primitive processing
  - The geometry shader can produce variable-length output!
  - 1 primitive in, *k* prims out
  - Is optional (not necessarily present on all GPUs)
- Note on the side: stream out
  - New, fixed-function
  - Divert primitive data to buffers
  - Can be transferred back to the OpenGL prog ("Transform Feedback")





(x,y,z)

(x',y',z')

attribute → Vertex Shader → Rasterizer → Fragment Shader → Buffer Op. →

varying

varying

uniform

(x,y,z)

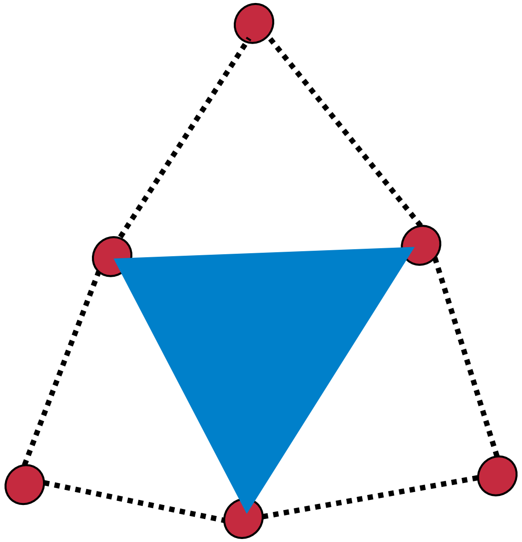
attribute → Vertex Shader → Geometry Shader → Rasterizer → Fragment Shader →

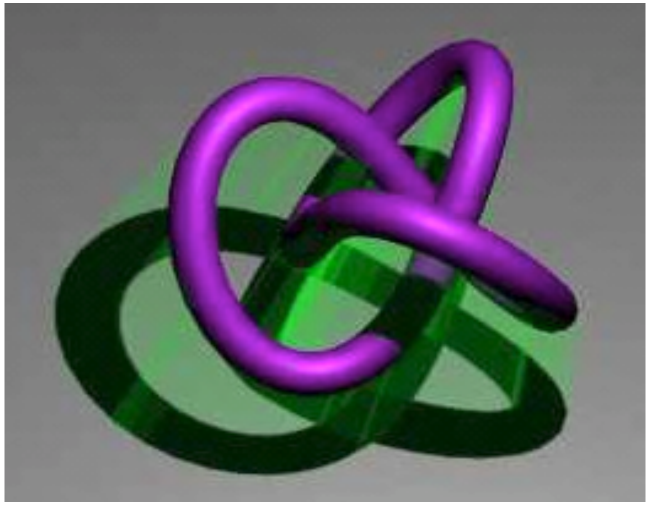
varying in

varying out

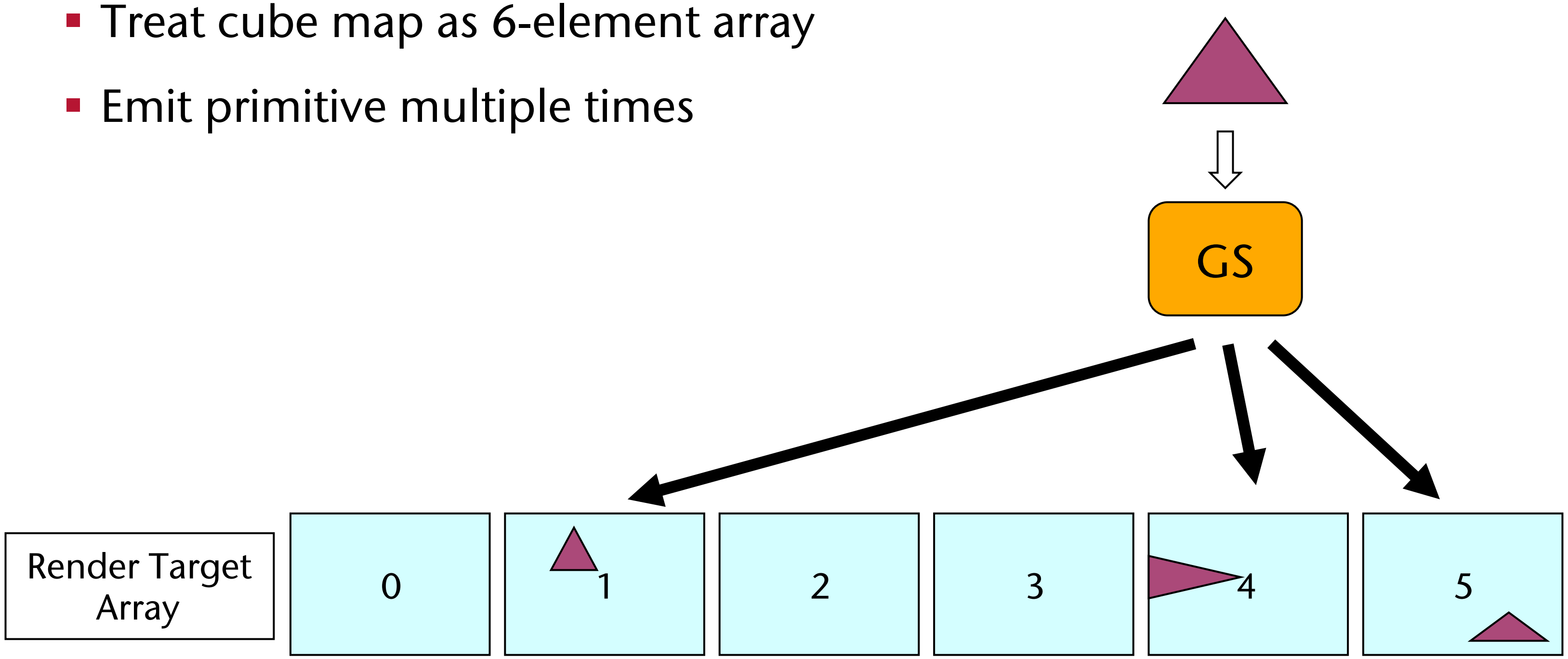
uniform

- The geometry shader's principle function:
  - In general "amplify geometry"
  - More precisely: can create or destroy primitives *on the GPU*
  - Entire primitive as input (optionally with adjacency)
  - Outputs zero or more primitives
    - 1024 scalars out max
- Example application:
  - Silhouette extrusion for shadow volumes



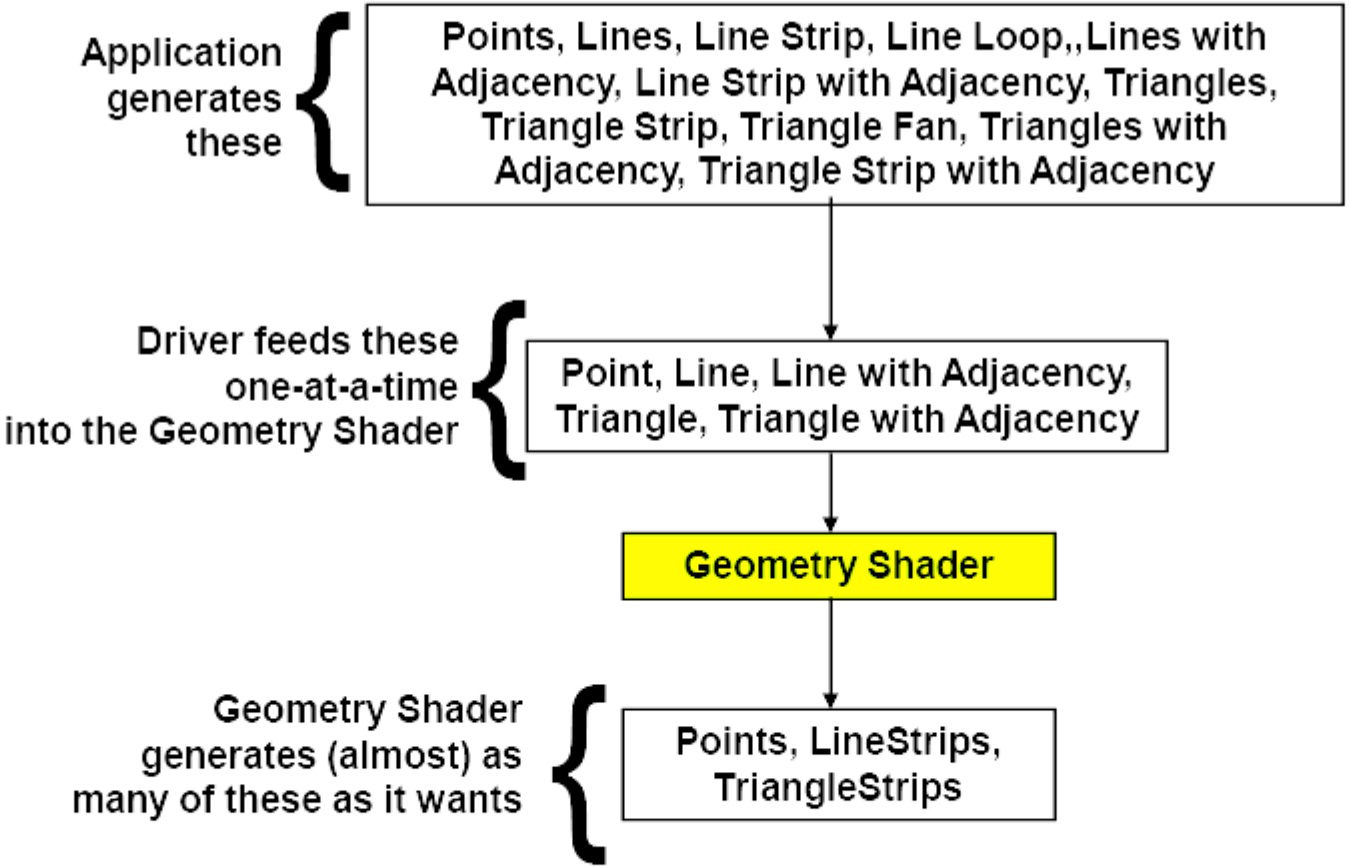


- Another feature of geometry shaders: can render the same geometry to multiple targets
- E.g., render to cube map in a single pass:
  - Treat cube map as 6-element array
  - Emit primitive multiple times



Some More Technical Details

- Input / output:



Application generates these

Points, Lines, Line Strip, Line Loop,,Lines with Adjacency, Line Strip with Adjacency, Triangles, Triangle Strip, Triangle Fan, Triangles with Adjacency, Triangle Strip with Adjacency

Driver feeds these one-at-a-time into the Geometry Shader

Point, Line, Line with Adjacency, Triangle, Triangle with Adjacency

Geometry Shader

Geometry Shader generates (almost) as many of these as it wants

Points, LineStrips, TriangleStrips

- In general, you must specify the type of the primitives input and output to and from the geometry shader
  - These need not necessarily be the same type
- Input type:

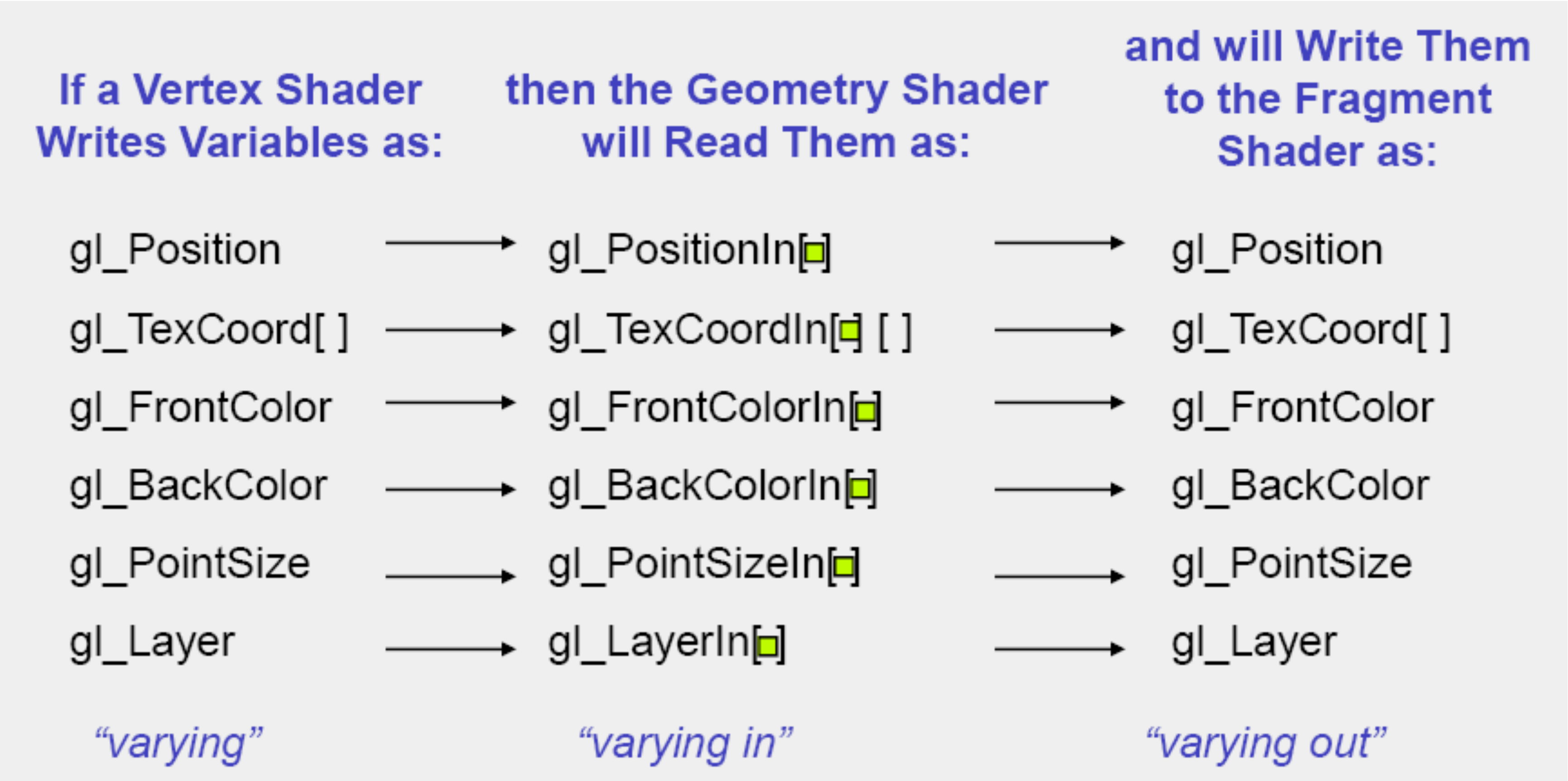
```
glProgramParameteri( shader_prog_name,
                     GL_GEOMETRY_INPUT_TYPE, int value );
```

  - **`value`** = primitive type that this geometry shader will be receiving
  - Possible values: GL_POINTS, GL_TRIANGLES, … (more later)
- Output type:

```
glProgramParameteri( shader_prog_name,
                     GL_GEOMETRY_OUTPUT_TYPE, int value );
```

  - **`value`** = primitive type that this geometry shader will output
  - Possible values: GL_POINTS, GL_LINE_STRIP, GL_TRIANGLES_STRIP

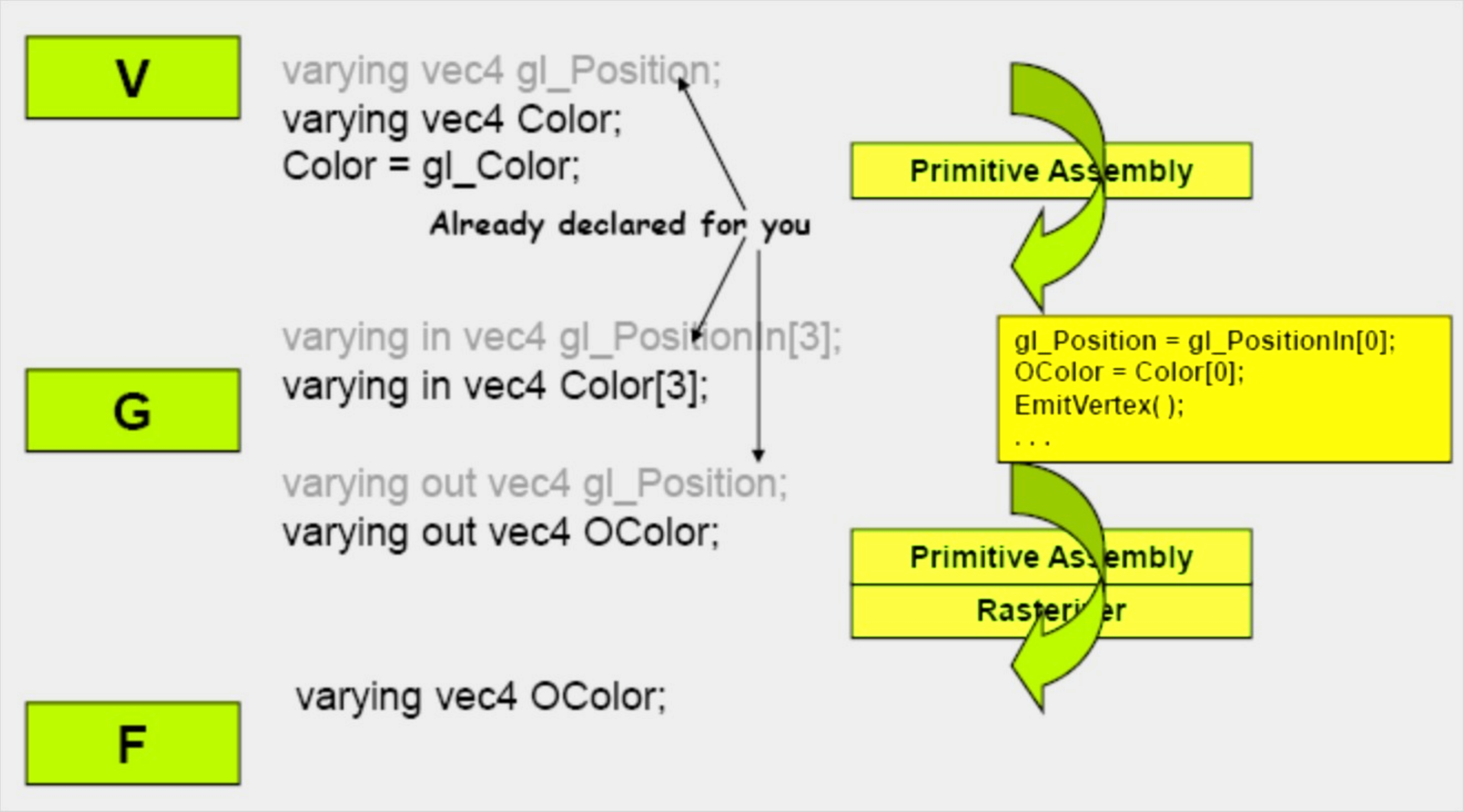
Data Flow of Varying the Principle Varying Variables



If a Vertex Shader Writes Variables as:		then the Geometry Shader will Read Them as:		and will Write Them to the Fragment Shader as:
gl_Position	→	gl_PositionIn[🟩]	→	gl_Position
gl_TexCoord[ ]	→	gl_TexCoordIn[🟩] [ ]	→	gl_TexCoord[ ]
gl_FrontColor	→	gl_FrontColorIn[🟩]	→	gl_FrontColor
gl_BackColor	→	gl_BackColorIn[🟩]	→	gl_BackColor
gl_PointSize	→	gl_PointSizeIn[🟩]	→	gl_PointSize
gl_Layer	→	gl_LayerIn[🟩]	→	gl_Layer
"varying"		*"varying in"*		*"varying out"*

🟩 *gl_VerticesIn*

- If a geometry shader is part of the shader program, then passing information from the vertex shader to the fragment shader can only happen via the geometry shader:



V

```
varying vec4 gl_Position;
varying vec4 Color;
Color = gl_Color;
```

Already declared for you

G

```
varying in vec4 gl_PositionIn[3];
varying in vec4 Color[3];

varying out vec4 gl_Position;
varying out vec4 OColor;
```

```
gl_Position = gl_PositionIn[0];
OColor = Color[0];
EmitVertex( );
. . .
```

F

```
varying vec4 OColor;
```

- Since you may not emit an unbounded number of points from a geometry shader, you are required to let OpenGL know the maximum number of points any instance of the shader will emit
- Set this parameter after creating the program, but before linking:

```
glProgramParameteri( shader_prog_name,
                     GL_GEOMETRY_VERTICES_OUT, int n );
```

- A few things you might trip over, when you try to write your first geometry shader:
  - It is an error to attach a geometry shader to a program without attaching a vertex shader
  - It is an error to use a geometry shader without specifying GL_GEOMETRY_VERTICES_OUT
  - The shader will not compile correctly without the #version and #extension pragmas

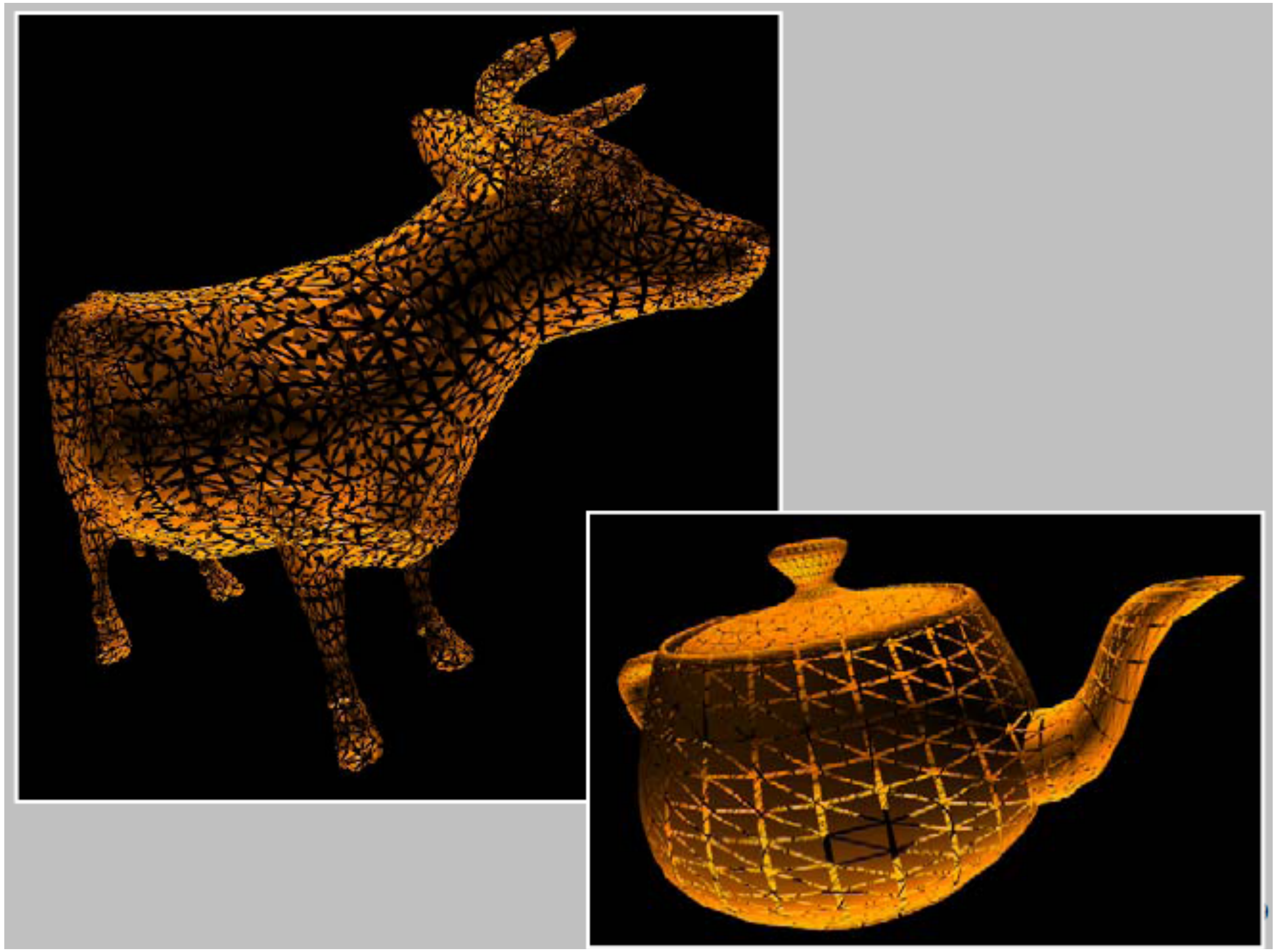
- The geometry shader generates geometry by repeatedly calling **`EmitVertex()`** and **`EndPrimitive()`**
- Note: there is no BeginPrimitive( ) routine. It is implied by
  - the start of the Geometry Shader, or
  - returning from the **`EndPrimitive()`** call

A Very Simple Geometry Shader Program

```
#version 120
#extension GL_EXT_geometry_shader4 : enable void
main(void)
{
   gl_Position = gl_PositionIn[0] + vec4(0.0, 0.04, 0.0, 0.0);
   gl_FrontColor = vec4(1.0, 0.0, 0.0, 1.0);
   EmitVertex();
   gl_Position = gl_PositionIn[0] + vec4(0.04, -0.04, 0.0, 0.0);
   gl_FrontColor = vec4(0.0, 1.0, 0.0, 1.0);
   EmitVertex();
   gl_Position = gl_PositionIn[0] + vec4(-0.04, -0.04, 0.0, 0.0);
   gl_FrontColor = vec4(0.0, 0.0, 1.0, 1.0);
   EmitVertex();
   EndPrimitive();
}
```

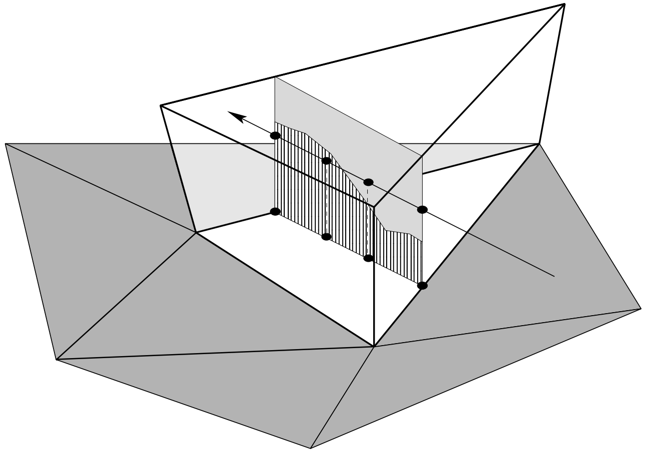
Examples

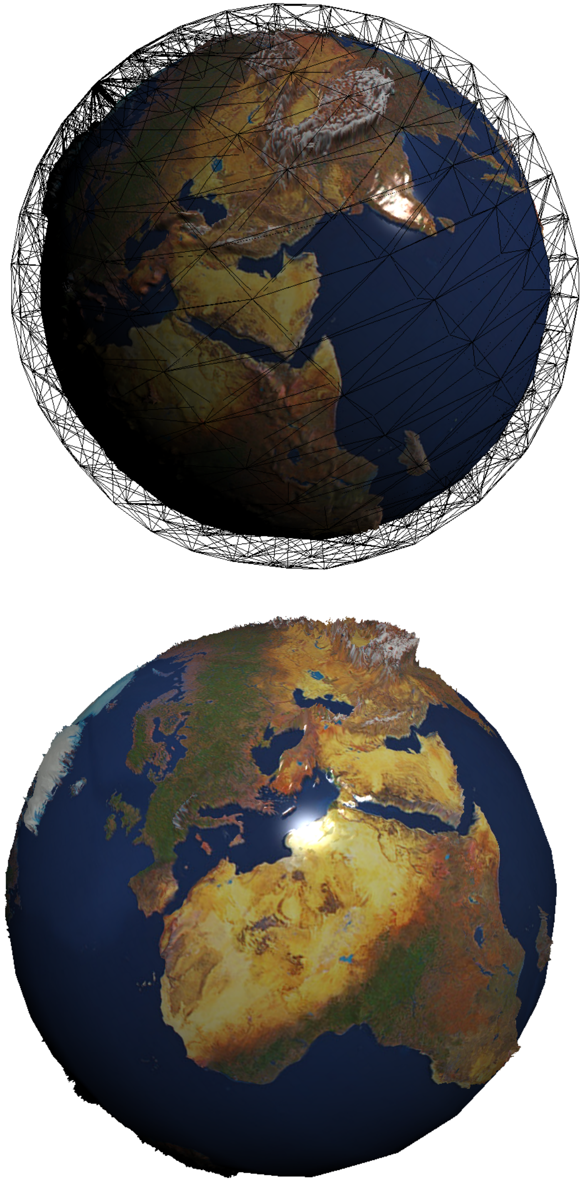
- Shrinking triangles:



Displacement Mapping

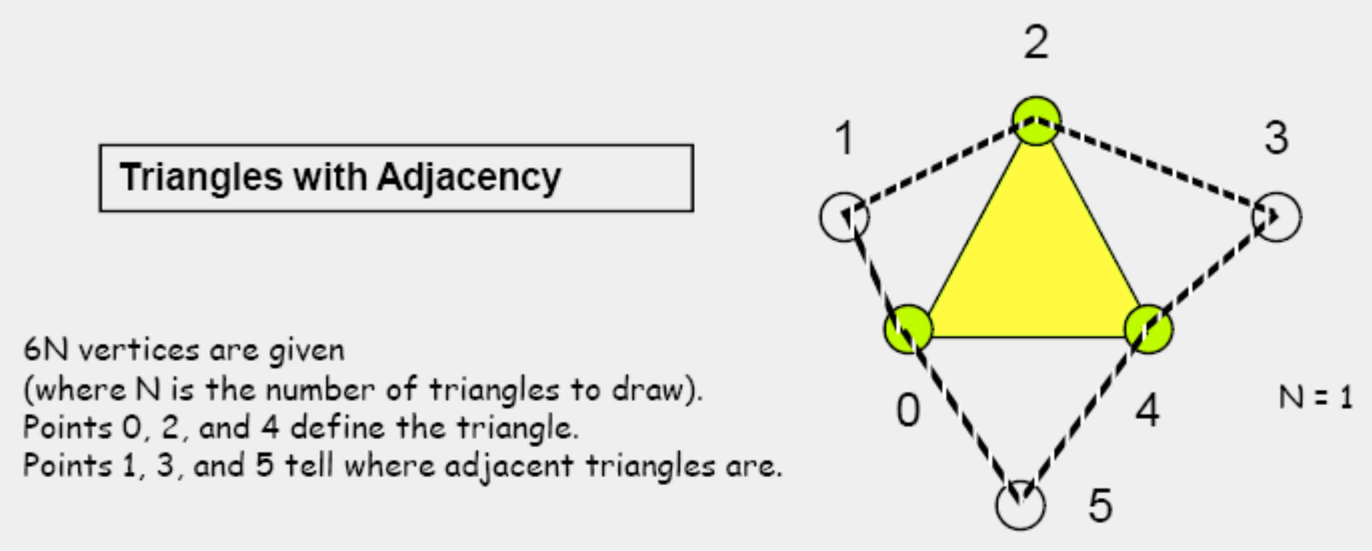
- Geometry shader extrudes prism at each face
- Fragment shader ray-casts against height field
- Shade or discard pixel depending on ray test





Intermezzo: Adjacency Information

- In addition to the conventional primitives (GL_TRIANGLE et al.), a few new primitives were introduced with geometry shaders
- The most frequent one: GL_TRIANGLES_WITH_ADJACENCY

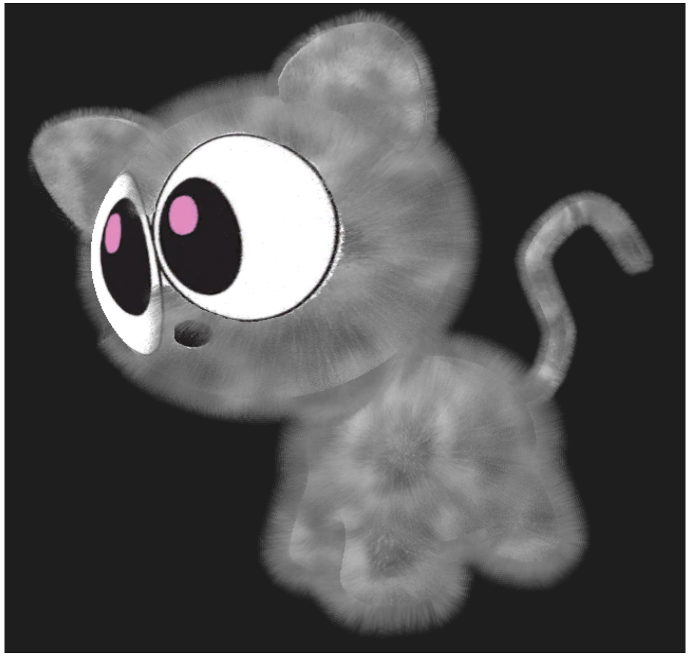


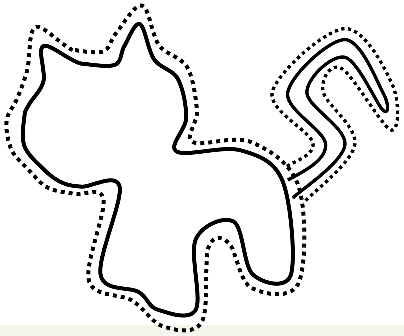
Triangles with Adjacency

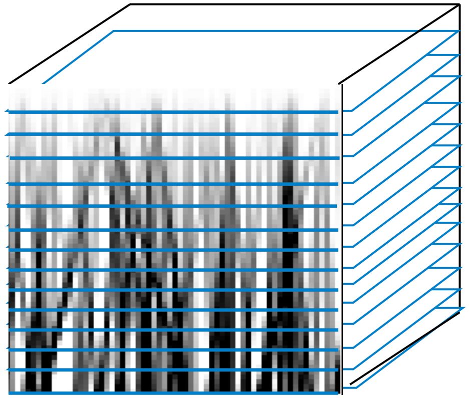
6N vertices are given
(where N is the number of triangles to draw).
Points 0, 2, and 4 define the triangle.
Points 1, 3, and 5 tell where adjacent triangles are.

Shells & Fins

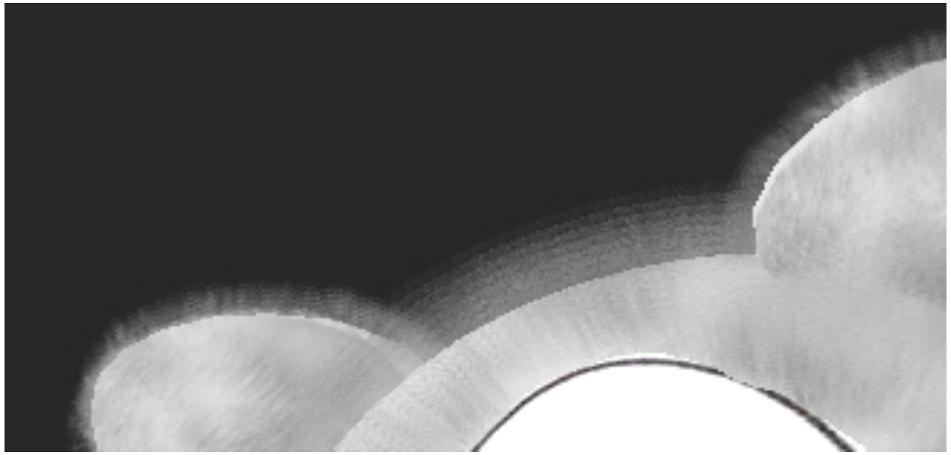
- Suppose, we want to generate a "fluffy", ghostly character like this
- Idea:
  - Render several shells (offset surfaces) around the original polygonal geometry
    - Can be done easily using the vertex shader
  - Put different textures on each shell the generate a volumetric, yet "gaseous" shell appearance





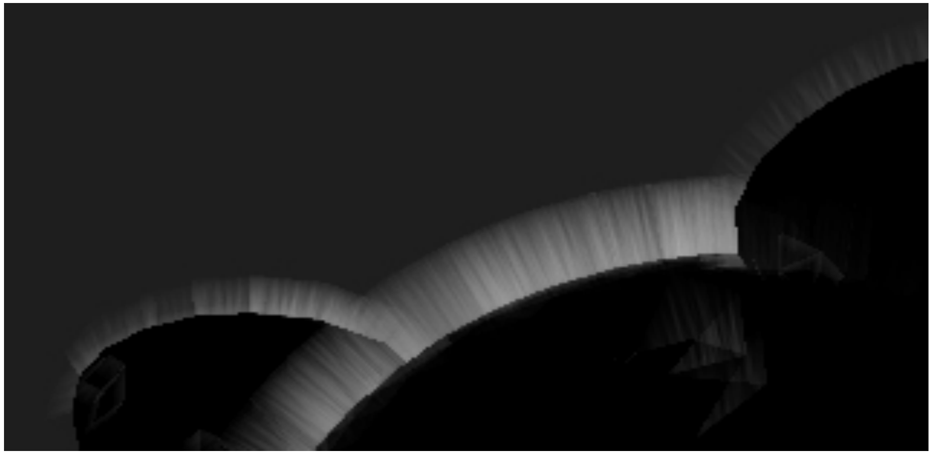


- Problem at the silhouettes:
- Solution: add "fins" at the silhouette
  - Fin = polygon standing on the edge between 2 silhouette polygons
- Makes problem much less noticeable



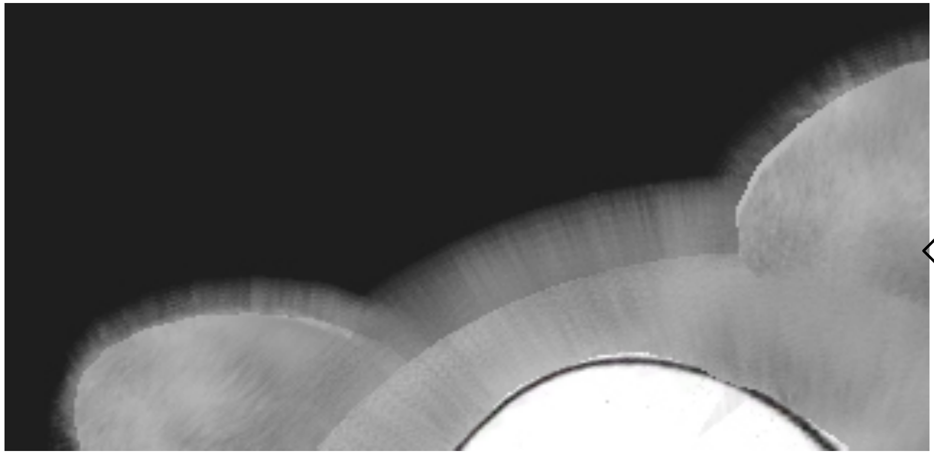
8 shells

+



fins

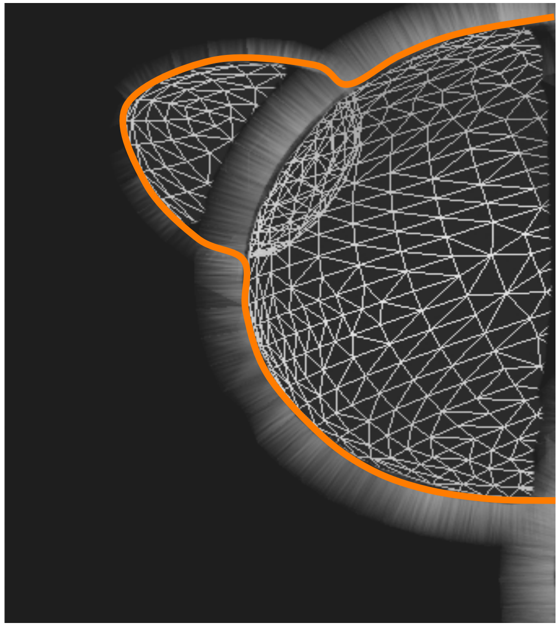
⇦

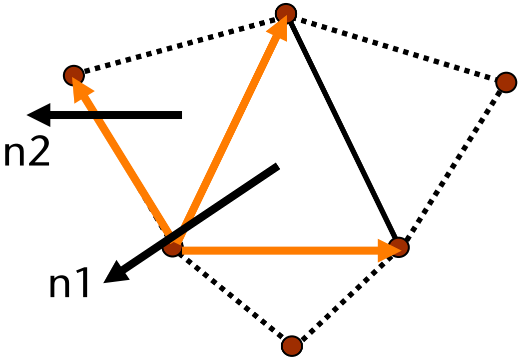


- Idea: fins can be generated in the geometry shader
- How it works:
    - All geometry goes through the geometry shader
    - Geometry shader checks whether or not the polygon has a silhouette edge:

      $$\text{silhouette} \Leftrightarrow \mathbf{en}_1 > 0 \;\wedge\; \mathbf{en}_2 < 0 \;\text{ or }\; \mathbf{en}_1 < 0 \;\wedge\; \mathbf{en}_2 > 0$$

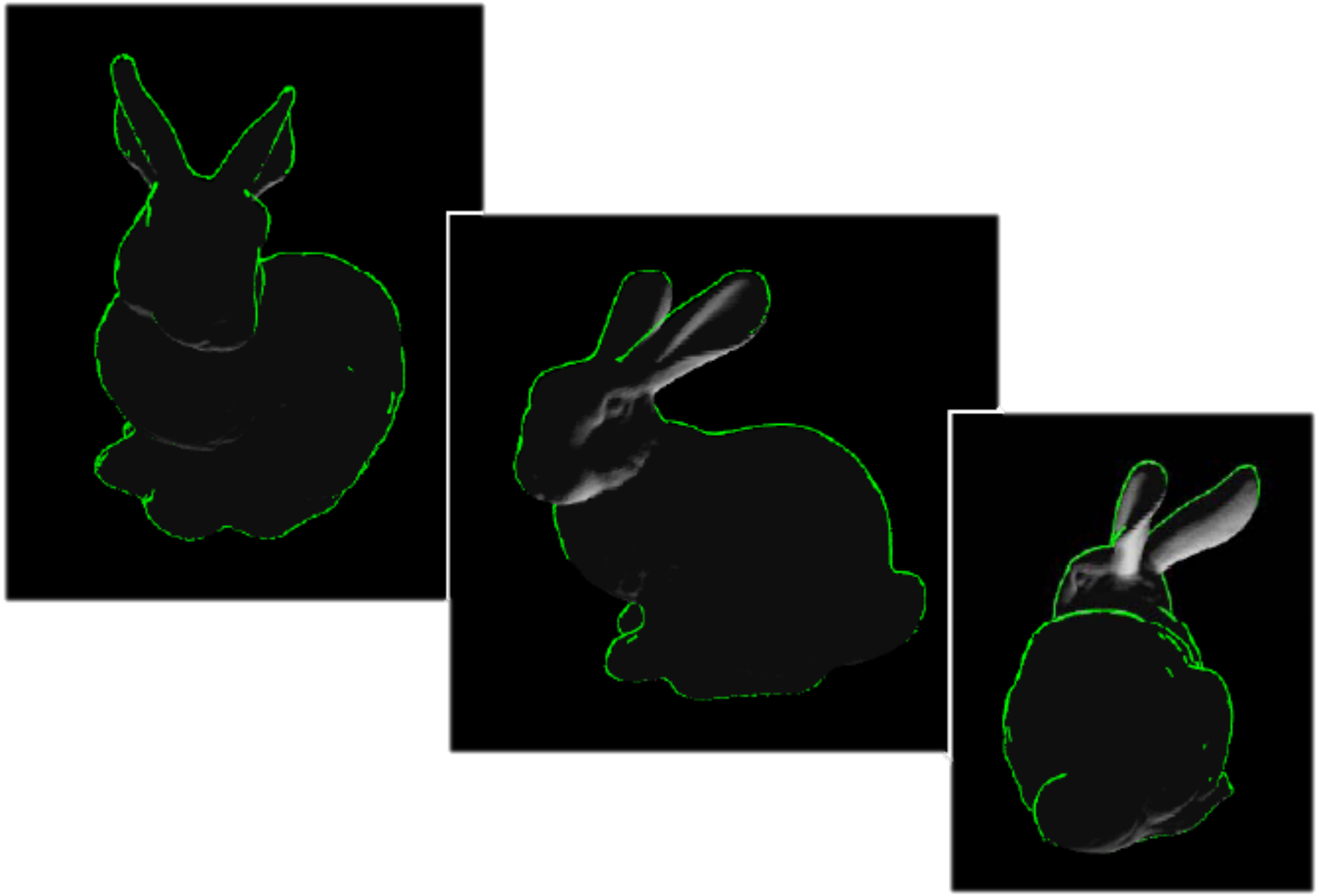
      where **e** = eye vector
    - If one edge = silhouette, then the geometry shader emits a fin polygon, and the input polygon
    - Else, it just emits the input polygon





Silhouette Rendering

- Goal:



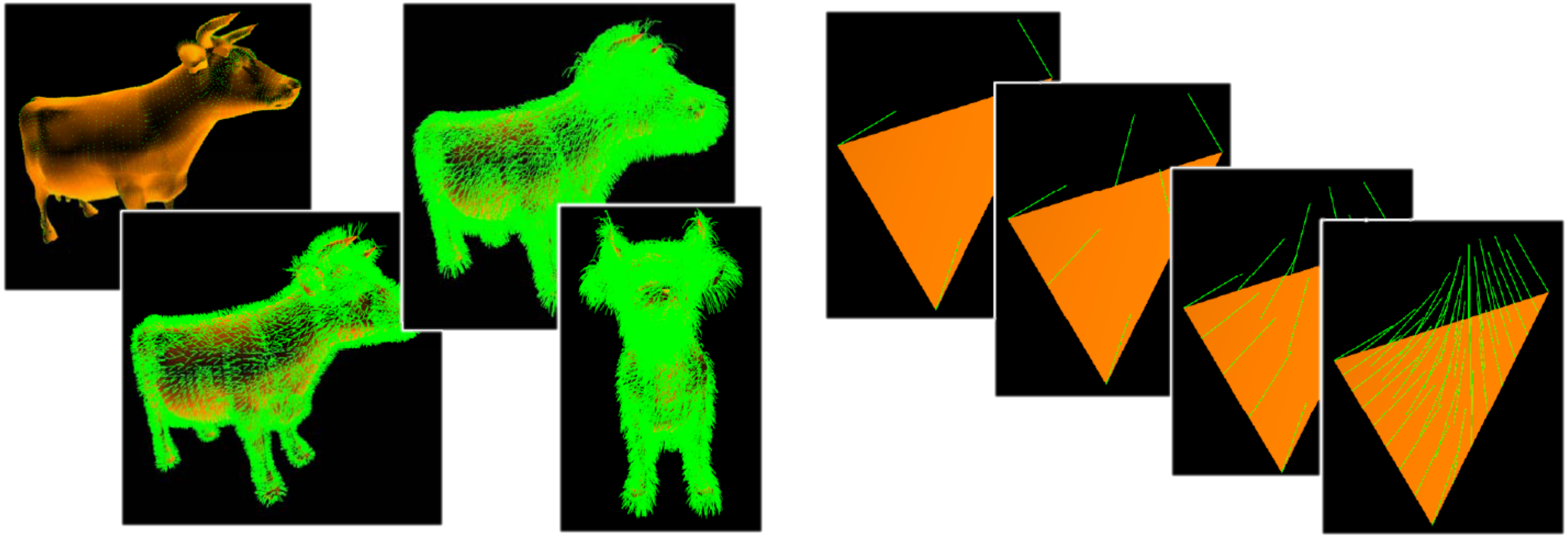
- Technique: 2-pass rendering

1. Pass: render geometry regularly
2. Pass: switch on geometry shader for silhouette rendering
    - Switch to green color for all geometry (no lighting)
    - Render geometry again
    - Input of geometry shader = triangles
    - Output = lines
    - Geometry shader checks, whether triangle contains silhouette edge
    - If yes ⟶ output line
    - If no ⟶ output no geometry

- Geometry shader input = GL_TRIANGLE_WITH_ADJACENCY
  output = GL_LINE_STRIP

More Applications of Geometry Shaders

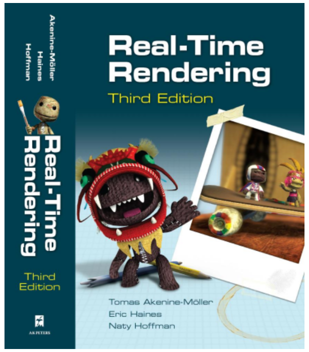
- Hedgehog Plots:



Shader Trees

Resources on Shaders

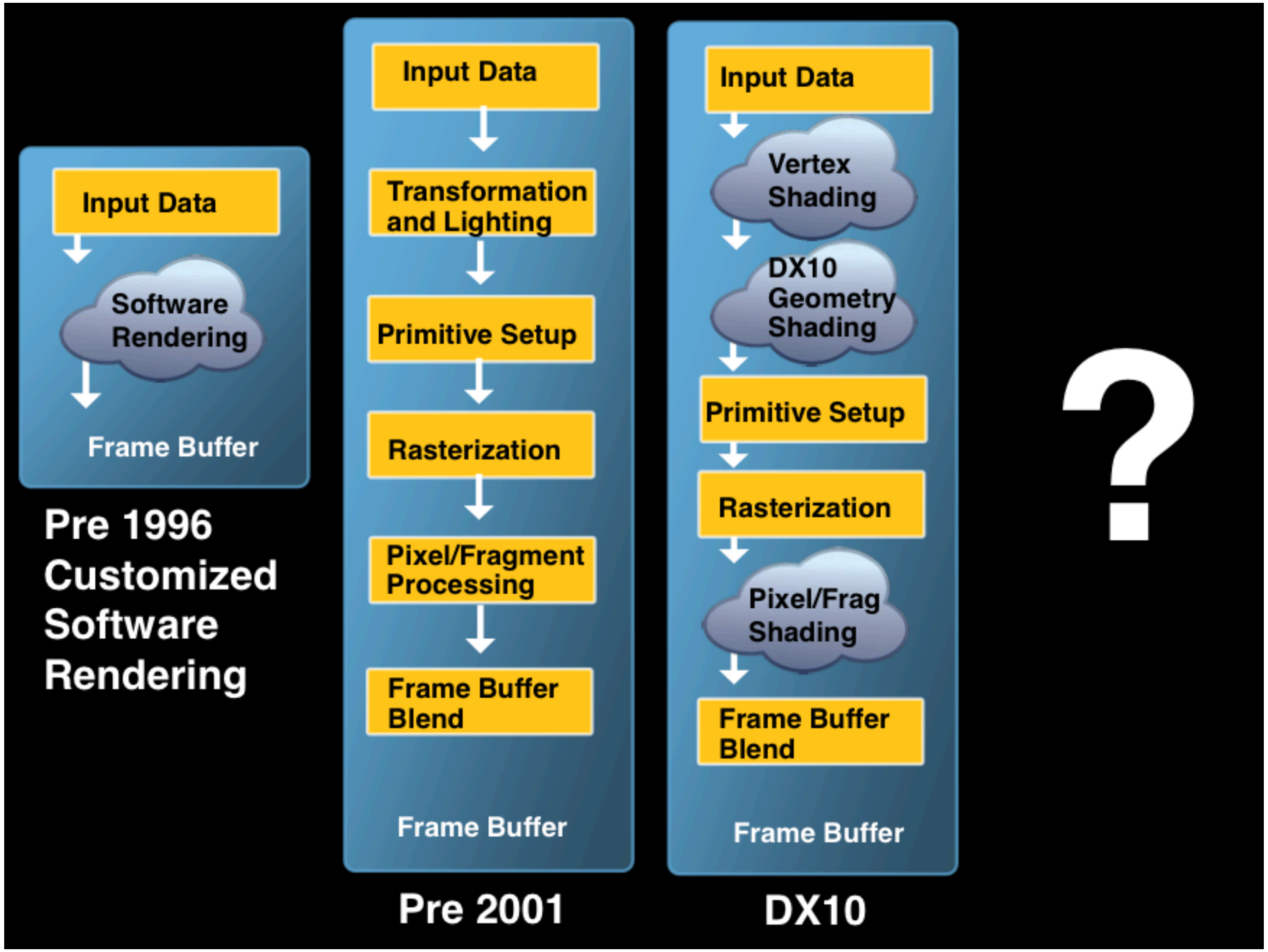
- Real-Time Rendering; 3rd edition

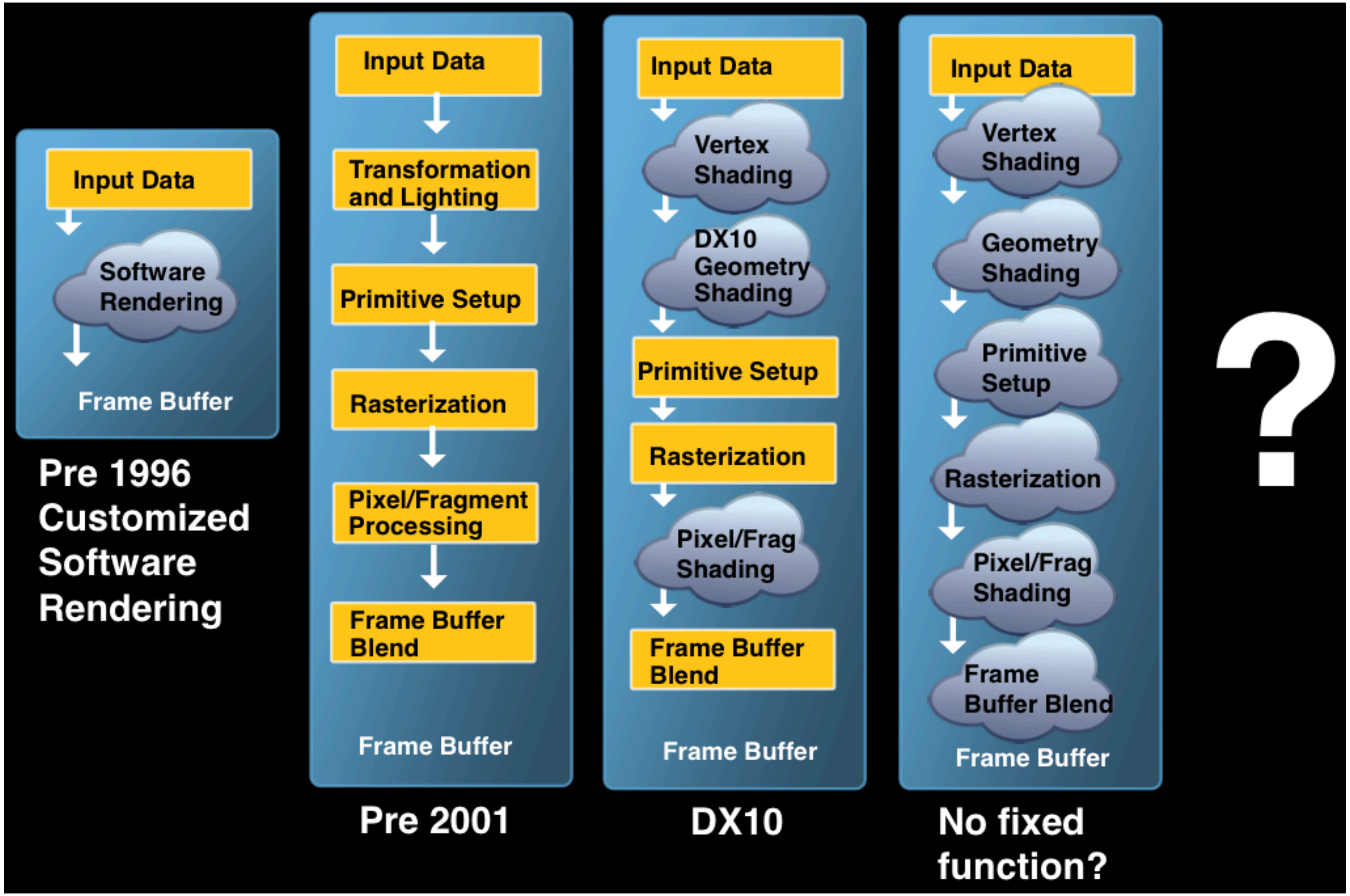


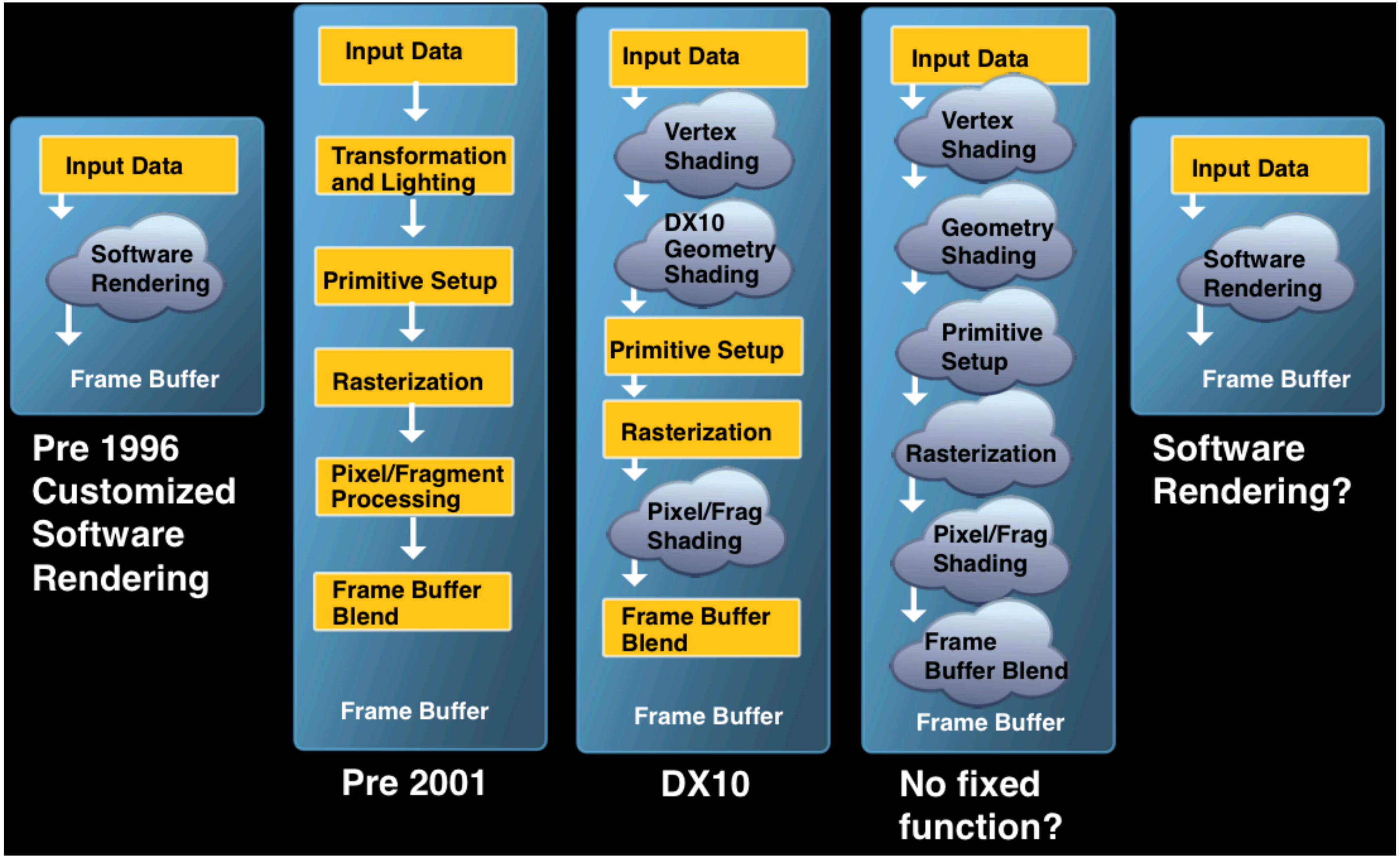
- Nvidia GPU Programming Guide:
  developer.nvidia.com/object/gpu_programming_guide.html

- On the geometry shader in particular:
  www.opengl.org/registry/specs/ARB/geometry_shader4.txt

The Future of GPUs?




Input Data
Software Rendering
Frame Buffer
Pre 1996 Customized Software Rendering
Input Data
Transformation and Lighting
Primitive Setup
Rasterization
Pixel/Fragment Processing
Frame Buffer Blend
Frame Buffer
Pre 2001
Input Data
Vertex Shading
DX10 Geometry Shading
Primitive Setup
Rasterization
Pixel/Frag Shading
Frame Buffer Blend
Frame Buffer
DX10
Input Data
Vertex Shading
Geometry Shading
Primitive Setup
Rasterization
Pixel/Frag Shading
Frame Buffer Blend
Frame Buffer
No fixed function?
?



Input Data

Software Rendering

Frame Buffer

Pre 1996 Customized Software Rendering

Input Data

Transformation and Lighting

Primitive Setup

Rasterization

Pixel/Fragment Processing

Frame Buffer Blend

Frame Buffer

Pre 2001

Input Data

Vertex Shading

DX10 Geometry Shading

Primitive Setup

Rasterization

Pixel/Frag Shading

Frame Buffer Blend

Frame Buffer

DX10

Input Data

Vertex Shading

Geometry Shading

Primitive Setup

Rasterization

Pixel/Frag Shading

Frame Buffer Blend

Frame Buffer

No fixed function?

Input Data

Software Rendering

Frame Buffer

Software Rendering?