Advanced Computer Graphics
Advanced Shader Programming

G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de
Recap

- Heute: programmierbare *vertex und fragment processors*
- Texturspeicher = allgemeiner Speicher für beliebige Daten
- Balancierung der Pipeline ist jetzt Programmierer's Aufgabe
More Abstract Overview of the Programmable Pipeline

- Vertices (in model coord)
  - glVertexAttribPointer
  - glUniform...

- Vertices (in view coord)
  - Uniforms
  - OpenGL State
  - Connectivity
  - GLuint
  - glDrawArrays() / glDrawElements() / Primitives

- Primitives
  - Rasterization

- Fragment Shader

- Fragment/Framebuffer Tests & Operations
  - GlFramebuffer

- Framebuffer

- New Fragmente

- Primitives

More Versatile Texturing by Shader Programming

- Declare texture in the shader (vertex or fragment):

  ```
  uniform sampler2D myTex;
  ```

- Load and bind texture in OpenGL program as usual:

  ```
  glBindTexture( GL_TEXTURE_2D, myTexture );
  glTexImage2D(...);
  ```

- Establish a connection between the two:

  ```
  uint mytex = glGetUniformLocation( prog, "myTex" );
  glUniform1i( mytex, 0 ); // 0 = texture unit, not ID
  ```

- Access in fragment shader:

  ```
  vec4 c = texture2D( myTex, gl_TexCoord[0].st );
  ```
Example: A Simple "Gloss" Texture

- Idea: expand the conventional Phong lighting by introducing a *specular reflection coefficient* that is mapped from a *texture* on the surface

\[
I_{\text{out}} = (r_d \cos \Phi + r_s \cos^p \Theta) \cdot I_{\text{in}}
\]

\[
r_s = r_s(u, v)
\]
• Goal: Brick texture

• Simplification & parameters:

- BrickStepSize.x
- BrickPercent.x
- BrickStepSize.y
- BrickPercent.y
- BrickColor
- MortarColor
Overview of Approach

- Vertex shader: normal lighting calculation
- Fragment shader:
  - For each fragment, determine if the point lies in the brick or in the mortar on the basis of the x/y coordinates of the corresponding point in objects space (!)
  - After that, multiply the corresponding color with intensity from lighting model
- First three steps towards a complete shader program:

brick1.frag  brick2  brick3
Noise

- Most procedural textures look too "clean"
  - Real objects show signs of dirt, grime, dents, random irregularities, etc.
  - Idea: add all sorts of noise
Ideal qualities of a noise function $f$

- At least $C^2$-continuous
- It’s sufficient, if it looks random
- No obvious patterns or repetitions
- Repeatable (same output with the same input)
- Convenient range, e.g. $[-1,1]$
- Can be defined for 1,...,4 dimensions
- Isotropic (invariant under rotation)
Why we don't just use a noise texture

Sphere rendered with a 3D texture to provide the noise. Notice the artifacts from linear interpolation.

Sphere rendered with procedural noise.
Simple Idea: Value Noise

1. Choose random y-values from [-1,1] at the integer positions

2. Interpolate in-between, e.g. cubically (linearly isn’t sufficient)
3. Generate multiple noise functions with different frequencies

4. Add them all up: produces noise at different "scales"
• **Persistence** = successive scaling of amplitude on successive octaves

\[
\text{perlin}(x) = \sum_{i=0}^{\infty} p^i n_i(2^i x), \quad x \in [0, 1], \quad p \in [0, 1]
\]

† Scaling along \(x\) for octaves

**Persistence**

• Example:

• Persistence = \(\frac{1}{2}\) → **pink noise**, persistence = 1 → **white noise**
• Same thing in 2D:

• Straight-forward generalization to higher dimensions
Gradient Noise

• Specify the gradients, instead of values, at integer x-points

• Interpolate to obtain values $y = f(x)$:
  
  • At position $x$, calculate $y_0$ and $y_1$ as values of the lines through $x=0$ and $x=1$ with the previously specified (random) gradients
  
  • Interpolate $y_0$ and $y_1$ with a sinusoidal 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: \( q''(0) = q''(1) \)
\( \rightarrow \) the entire noise function is \( C^2 \)-continuous

• Example where one can easily see this:
Gradient Noise in 2D, \( f : \mathbb{R}^2 \rightarrow \mathbb{R} \)

- Set gradients at integer grid points
  - \( \text{Gradient} = 2D \) vector (not necessarily of length 1)
- Interpolation (as in 1D):
  - Wlog., at \( P = (x,y) \in [0,1] \times [0,1] \)
  - 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 \( P = (x,y) \):
    \[
    z_{00} = g_{00} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \quad z_{10} = g_{10} \cdot \begin{pmatrix} x - 1 \\ y \end{pmatrix} \\
    z_{01} = g_{01} \cdot \begin{pmatrix} x \\ y - 1 \end{pmatrix} \quad z_{11} = g_{11} \cdot \begin{pmatrix} x - 1 \\ y - 1 \end{pmatrix}
    \]
• Blending of 4 z-values through bilinear interpolation:

\[ z_{x0} = (1 - q(x))z_{00} + q(x)z_{10} , \quad z_{x1} = (1 - q(x))z_{01} + q(x)z_{11} \]

\[ z_{xy} = (1 - q(y))z_{x0} + q(y)z_{x1} \]

• 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 as weights

• And in \(d\)-dim. space? \(\rightarrow\) complexity is \(O(d^2 \cdot 2^d)\)!
Simplex Noise

• The *d*-dimensional simplex := barycentre combination of \( d+1 \) affinely independent points

• Examples:
  • 1D simplex = line, 2D simplex = triangle, 3D simplex = tetrahedron

• In general:
  • Points \( P_0, ..., P_d \) are given
  • \( d \)-dim. simplex = all points \( X \) with
    \[
    X = P_0 + \sum_{i=1}^{d} s_i u_i
    \]
    with
    \[
    u_i = P_i - P_0 , \ s_i \geq 0 , \ \sum_{i=0}^{d} s_i \leq 1
    \]
Simplicial Tessellation

- In general, the following is true: it is possible to partition $d$-dimensional space \((tessellation)\) with \textit{equilateral} $d$-dimensional simplices.
- Using equilateral $d$-dimensional simplices, one can partition a cube that was suitably "compressed" along one of its diagonals.

\begin{itemize}
  \item Such a "compressed" $d$-dimensional cube contains $d!$ many simplices.
\end{itemize}
Construction of the Noise Function

- Given: a simplex tessellation (hence "simplex noise") and gradients at each node/vertex

1. Determine the simplex in which a query point \( P \) lies
2. Determine all of its vertices and the gradients there
3. Determine (as before) the value of these "gradient ramps" in query point \( P \)
4. Generate a weighted sum of these values

- Choose weighting functions so that the “influence” of a simplex grid point only extends to its incident simplices
• Advantage: has only complexity $O(d)$
• For details see "Simplex noise demystified" (on the homepage of this course)
• Comparison between classical value noise and simplex noise:
• Four noise functions are defined in the GLSL standard:
  \[ \text{float } \text{noise1(gentype)}, \text{ vec2 } \text{noise2(gentype)}, \]
  \[ \text{vec3 } \text{noise3(gentype)}, \text{ vec4 } \text{noise4(gentype)}. \]

• Calling such a noise function:
  \[ v = \text{noise2}( f \times x + t, f \times y + t ) \]

  • With \( f \), one can control the spatial frequency;
    with \( t \), one can generate a shifting animation (\( t="\text{time}"\)).

• Analogous for 1D and 3D noise

• Caution: range is \([-1,+1]\]!

• Cons: are not implemented everywhere
  • Often very slowooooooow ...
Example (cont'd): Application of Noise to our Procedural Brick Texture

4. Color var. (low freq.)
5. High-freq. variations
6. Black spots
7. Curvy brick edges

The code for this example is on the course's homepage (after unpacking the archive, it is in directory `vorlesung_demos files brick.vert and brick[4-7].frag`)
Example in Movies

If you look closely, you can see yellowed printed material, little scratches, and paint worn off metal surfaces that tell you this collection is old but well-maintained.
Other Examples for the Applications of Noise

Ken Perlin's famous solid textured marble vase, 1985

Procedural bump mapping, done by computing noise in the pixel shader and using that for perturbing the surface normal
\[ g = a \times \text{perlin}(x, y, z) \]
\[ \text{grain} = g - \text{int}(g) \]
Warped Noise

• Sometimes, you need different frequencies in different parts of the noise image/domain

• General idea: perform image warping

• Method:
  • Define polylines where noise image should "contract" (increases frequency)
  • Use generalized barycentric coordinates to move original pixels closer to "borders"
Digression on Randomness

Michael Noll, 1962:
Computer Composition with Lines
Based on Piet Mondrian’s
Composition with Lines

Piet Mondrian, 1917
Random-dot stereograms. 3D object hidden in random images.
[Bela Julesz, Hungarian psychologist]
Is it random? What makes it random?
Is it random? What makes it random?
Is Your Random Number Generator Random?

- Spectral test:
  - Calculate $x_i = \text{rand}()$, for $i = 1, \ldots, N$ ($N$ large)
  - Create 2D scatter plot of all points ($x_i, x_{i+1}$)
    - Or in 3D using ($x_i, x_{i+1}, x_{i+2}$)

Apple's `rand()` function in C

IBM's RANDU function
Eudaemons = Name of a group of physics graduates from University of Santa Cruz who understood that roulette wheels obey Newtonian physics, but is just very sensitive to initial conditions. Using miniaturized computers, hidden in special shoes, they could capture the state of the ball and the wheel, and could increase their odds by 44%.

[Thomas A. Bass : The Eudaemonic Pie, 1985]
Ambient Occlusion

• Motivation:
  • Remember the rendering equation
    \[ L_r(x, \omega_r) = \int_\Omega \rho(x, \omega_r, \omega_i) L_i(x, \omega_i) \cos(\theta_i) d\omega_i \]

• Assume that \( \rho \) and incoming light is constant from every direction
  \[ \rightarrow \text{ambient occlusion:} \]
  \[ A(x) = \int_\Omega v(x, \omega_i) n \cdot \omega_i d\omega_i \]

where \( v(x, \omega) = \{0, 1\} = \text{visibility in direction } \omega \)
• Further simplification: only check for self-occlusion → object-space ambient occlusion
• Can be pre-computed per object as kind of a "light map"
  • Independent of light direction and other objects in scene
  • Can be multiplied with texture at run-time per fragment
Ambient Occlusion Effect Depends on Length of Occlusion Feelers

(a) Ray length: 160 units  (b) Ray length: 40 units  (c) Ray length: 10 units  (d) Ray length: 2 units
Screen-Space Ambient Occlusion

- Principle idea: sample neighborhood around each point on surface, calculate ambient occlusion term, use in lighting model as a factor
- One solution: use deferred shading, i.e., render into G-buffer in first pass
  - In second pass, for each receiver, use z-buffer to check visibility (wrt. viewpoint!) of samples around receiver, approximate occlusion term $A \approx \frac{1}{\text{# samples}} \sum_{\text{green}} v_i \cdot n \cdot \omega_i$, then evaluate lighting model
Method 2

- Re-project every pixel in frame buffer to the 3D point (in camera space!) → receiver point \( p \)
- Convert neighborhood radius \( R \) to radius on screen, consider all pixels within, reproject → sample points \( \mathbf{q}_i \)
- Approximate surface around sample points by disks w/ radius \( r_i \), oriented towards receiver
- Project onto hemisphere around receiver
- Accumulate

\[
A \approx \sum_{|\mathbf{q}_i - \mathbf{p}| < R} F'_i \mathbf{n} \cdot (\mathbf{q}_i - \mathbf{p}) = \sum_{|\mathbf{q}_i - \mathbf{p}| < R} \frac{F_i}{d_i^2} \mathbf{n} \cdot (\mathbf{q}_i - \mathbf{p})
\]
Experiment!

- Change parameters: size of neighborhood $R$, radius of disks $r_i$
- Specifically for method 2:
  - Place spheres around samples? (project these onto hemi-sphere of receiver point)
  - Should radius of sample disks/spheres depend on distance $d_i$?
  - Compute area of spherical cap on hemi-sphere covered by sample disks (don't just scale area $F_i$ of the disks)
  - Orient disks perpendicular to normal in sample points, not towards receiver pt?
  - How to account for parts of surface around $p$ that are visible from $p$, but invisible from viewpoint?
- Does method 1 or method 2 produce more aesthetically pleasing results?
Results
Effects Appear Usually Exaggerated Compared to Global Illumination
Light Refraction

• With shaders, one can implement simple approximations of global effects
• Example: light refraction
• What do we need to calculate the refracted ray?
  • Snell's Law: \( n_1 \sin \theta_1 = n_2 \sin \theta_2 \)
  • Needed: \( \mathbf{n}, \mathbf{v}, n_1, n_2 \)
    • Everything is available in the fragment shader!
    • So, one can calculate \( t \) \textit{per pixel}
• So why is rendering transparent objs difficult?
  • In order to calculate the correct intersection points of the refracted ray, one needs the entire geometry!
Approximation: one entry point, one exit point in/out of transparent object

1. Step: determine the exit point

\[ P_2 = P_1 + d \mathbf{t} \]

- Idea: approximate \( d \)
- To do that, render a depth map of the back-facing polygons in the first pass, from the viewpoint
- Use binary search to find a good approximation of \( d \) (ca. 5 iterations suffice)
Details on the binary search for finding the distance between $P_1$ and $P_2$

- Situation: given a ray $t$, with $t_z < 0$, and two "bracket" points $A^{(0)}$ and $B^{(0)}$ (in camera space!), between which the intersection point must be; and a precomputed depth map
- Compute midpoint $M^{(0)}$
- Use $(M_x^{(0)}, M_y^{(0)})$ to index the depth map → $\tilde{d}$
- If $\tilde{d} > M_z^{(0)} \Rightarrow$ set $A^{(1)} = M^{(0)}$
- If $\tilde{d} < M_z^{(0)} \Rightarrow$ set $B^{(1)} = M^{(0)}$
- Repeat until convergence or max iterations
- Warning: drawing cuts a few corners! (for clarity)
2. Step: determine the normal in $P_2$
   - To do that, render a normal map of all back-facing polygons from the viewpoint (yet another pass before the actual rendering)
   - Project $P_2$ with respect to the viewpoint into screen space
   - Index the normal map

3. Step:
   - Determine $t_2$
   - Index an environment map
• Many open challenges:
  • When *depth complexity* > 2:
    • Which normal/which depth value should be stored in the depth/normal maps?
  • Approximation of distance
    • If object is highly non-convex, the approximation method can fail
  • Combination of reflected and refracted rays with Fresnel terms
  • Aliasing
Examples

With max. number of internal reflections

\textit{Our Method} \hspace{1cm} \textit{Ray Traced}

With different number of internal reflections

2 bounces
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, $k$ primitives out
  - Is optional (not necessarily present on all GPUs)
- Note on the side: features stream out
  - New, fixed function
  - Divert primitive data to buffers
  - Can be transferred back to the OpenGL program ("Transform Feedback")
Vertex Shader

uniform

attribute

(x, y, z)

Fragment Shader

Rasterizer

varying in


(x', y', z')

Geometry Shader

varying out

(x, y, z)

varying variables

uniform

attribute

(x, y, z)
Features / Purposes of the Geometry Shader

- The geometry shader's principle function:
  - In general "amplify geometry"
  - More precisely: can create (or destroy) primitives on the GPU
  - Input = one complete primitive (optionally with adjacency)
  - Output: zero or more primitives (max 1024)
- 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 primitives

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

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

  **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**

  **Geometry Shader**

  **Points, Line Strips, Triangle Strips**
In general, you must specify the type of the primitives that will be input and output to and from the geometry shader.

- These need not necessarily be the same type.

**Input type:**

```c
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:**

```c
glProgramParameteri( shader_prog_name,
                    GL_GEOMETRY_OUTPUT_TYPE, int value );
```
Data Flow of the Principle Predefined Varying Variables

If a Vertex Shader writes variables as:

- gl_Position
- gl_TexCoord[]
- gl_FrontColor
- gl_BackColor
- gl_PointSize
- gl_Layer

then the Geometry Shader will read them as:

- gl_PositionIn[]
- gl_TexCoordIn[]
- gl_FrontColorIn[]
- gl_BackColorIn[]
- gl_PointSizeIn[]
- gl_LayerIn[]

and will write them to the Fragment Shader as:

- gl_Position
- gl_TexCoord[]
- gl_FrontColor
- gl_BackColor
- gl_PointSize
- 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:

```
Vertex shader code
VColor = gl_Color;
```

```
Fragm. Shader
varying vec4 FColor;
Grey = already declared for you
```

```
Geom. Shader
varying in vec4 gl_PositionIn[3];
```

```
Vertex Shader
varying vec4 gl_Position;
```

```
Fragm. Shader
varying out vec4 gl_Position;
```

```
Vertex shader code
VColor = gl_Color;
```

```
Fragm. Shader
FColor = VColor[0];
```

```
Emitvertex();
```

```
gl_Position = gl_PositionIn[0];
FColor = VColor[0];
```

```
```
...
• 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:

```c
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 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 previous `EndPrimitive()` call.
A Very Simple Geometry Shader Program

```glsl
#version 120
#extension GL_EXT_geometry_shader4 : enable void

main(void)
{
    float d = 0.04;
    gl_Position = gl_PositionIn[0] + vec4(0.0, d, 0.0, 0.0);
    gl_FrontColor = vec4(1.0, 0.0, 0.0, 1.0);
    EmitVertex();
    gl_Position = gl_PositionIn[0] + vec4(d, -d, 0.0, 0.0);
    gl_FrontColor = vec4(0.0, 1.0, 0.0, 1.0);
    EmitVertex();
    gl_Position = gl_PositionIn[0] + vec4(-d, -d, 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_TRIANGLES et al.), a few new primitives were introduced with geometry shaders
- The most frequent one: GL_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", ghost-like 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 to 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
• 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:
    $$\text{silhouette } \iff \mathbf{e}_1 > 0 \land \mathbf{e}_2 < 0$$
    where \( \mathbf{e} = \) eye vector, from edge to eye
  • If edge = silhouette edge, then the geometry shader emits a fin polygon and the input polygon
  • Else, it just emits the input polygon
  • Do we need to check the "other orientation" of signs?
A simple way to implement fur is to draw many shells of an object. The shells are made up of vertices slid along the normals of the object. Each shell’s transparency is modulated with a noise texture on a per-fragment basis to approximate a random distribution of fur.

The vertex shader in this exhibit slides the vertices of the object along its normals, and computes lighting.

The fragment shader samples a color texture for the color of the surface and then uses a noise texture to modulate the alpha of fragments.
Example Application: Lost Planet Extreme Condition

No, they understand me.
More tricks are usually needed to make it look really good:

Texture for color
Texture for angle of fur hairs
Noise texture for length of fur hairs

Furthermore, one should try to render self-shadowing of strands of fur hairs...
Typically, what you as a programmer need to do is to write the shader and expose the parameters via a GUI to the artists, so they can determine the best look.
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
     - With tolerance to make silhouette thicker
     - If yes $\rightarrow$ output line
More Applications of Geometry Shaders

• Hedgehog Plots:
Concluding Demos

This is the Inferno shader outlined in the Orange Book.
Impossible Objects
The Future of GPUs?

Pre 1986: pure software rendering

Pre 2001: complete, efficient HW pipeline

OpenGL 3.x: first real programmability

No fixed function any more?

Pure software rendering?
Resources on Shaders

• Real-Time Rendering; 3rd edition

• The tutorial on this course's home page

• OpenGL Shading Language Reference: https://www.khronos.org/opengl/

• On the geometry shader in particular: https://www.khronos.org/opengl/wiki/Geometry_Shader