Lesson 9 – Tessellation shaders

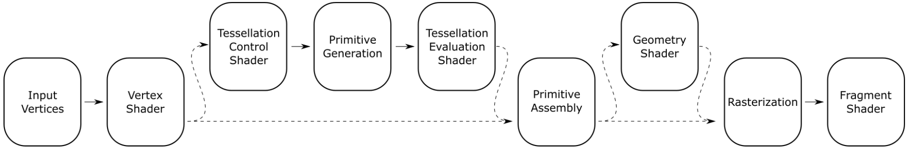
PV227 – GPU Rendering

Jiří Chmelík, Jan Čejka
Fakulta informatiky Masarykovy univerzity

12. 11. 2018

Tessellation Shaders

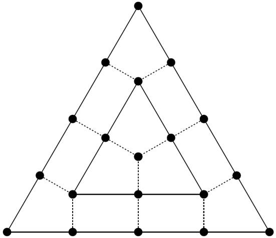
- new programmable stage (optional)
- between vertex shader and geometry shader,
- use the correct spelling :-)



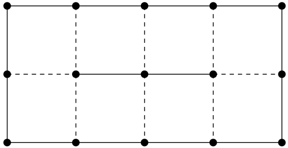
Tessellation Shaders

- Tessellation Control Shader (TCS)
  - Hull Shader in HLSL
  - optional, programmable
  - computes the parameters of the tessellation (the density of the mesh)
- Primitive generation
  - fixed
- Tessellation Evaluation Shader (TES)
  - Domain Shader in HLSL
  - required, programmable
  - computes the data of each generated vertex, like vertex shaders

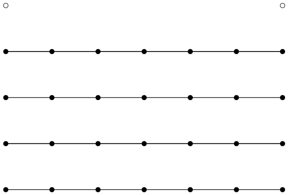
Primitive Generation



Triangle



Quad



Isolines

Patches

- New primitive, only for tessellation shaders
- Consist of 1 – 32 vertices
  glPatchParameteri(GL_PATCH_VERTICES, 16);
  glDrawArrays(GL_PATCHES, ...);
- Individual patches, no strips
- OpenGL does **not** define the mapping between input vertices and control points, the programmer does!

Tessellation Control Shader in GLSL

- Consumes one patch, generates one patch, like geometry shader
- Unlike geometry shaders, TCS is executed once per output vertex.
- Computes the following:
  - parameters of the tessellation
  - parameters of the whole patch
  - data of each patch control point.
- Number of generated control points (vertices)
  layout(vertices = 4) out;
- Index of the vertex for which this TCS is executed
  gl_InvocationID

Tessellation Control Shader in GLSL

- Parameters of the tessellation:
  - *gl_TessLevelInner[2]* describes the density inside the patch
  - *gl_TessLevelOuter[4]* describes the density at the boundary of the patch
  - When set to 0, the whole patch is discarded
- Per patch data, marked as *out patch*, passed into TES
  - Example: *out patch int materialIdx;*
- Usually computed only by one thread, e.g. by the thread with *gl_InvocationID* $=$ *0*;

Tessellation Control Shader in GLSL

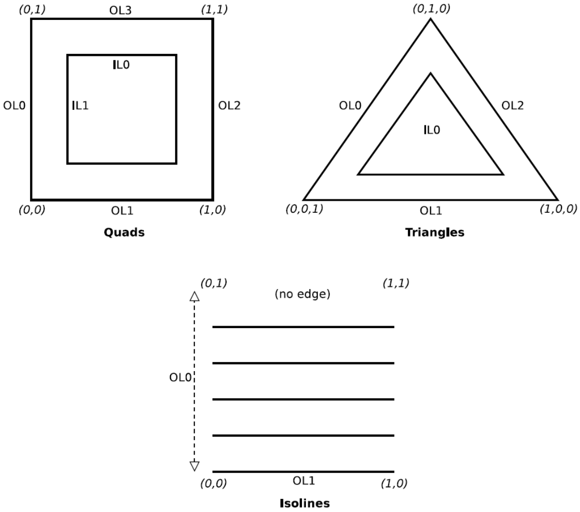
- Array of per vertex input data from the vertex shader
  - Example: *in vec4 position_vs[];*
  - Every TCS has access to each per vertex input data
- Array of per vertex output data into the TES
  - Example: *out vec4 position_tcs[];*
  - Every TCS has **readonly** access to each per vertex output data
  - TCS car write only the data of **its own** vertex
  - Use *barrier()* to make sure the data written by TCS are visible to other TCS.

- TCS is optional, when missing:
  - Per vertex data passes through from vertex shader into TES
  - The number of patch vertices stays the same
  - Tessellation levels defined from C++ code using *glPatchParameterfv*

Tessellation Evaluation Shader in GLSL

- Computes the data of each generated vertex
- Defines the patch topology: *layout(...) in;*
  - triangles / quads / isolines
  - fractional_odd_spacing / fractional_even_spacing / equal_spacing
  - cw / ccw
  - point_mode / (nothing)
  - Example: *layout(quads, equal_spacing, ccw, point_mode) in;*
- Array of per vertex input data from TCS: *in vec4 position_vs[];*
- Per patch data, from TCS: *in patch int materialIdx;*
- Coordinate of the tessellated vertex in the patch
  - *vec3 gl_TessCoord*
  - triangles uses 3 coordinates (xyz)
  - quads and isolines use 2 coordinates (xy)
- Output: like the output of vertex shader

Patch topology

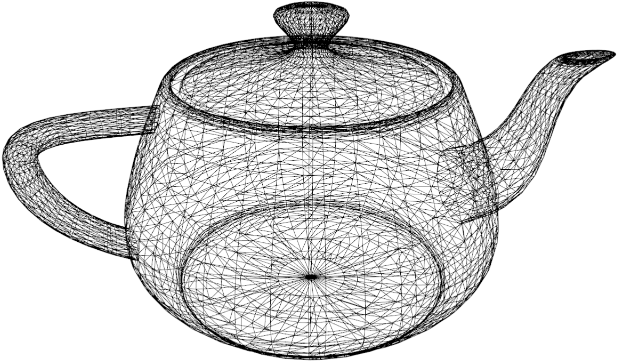


Task: Examine patch topology

- **Task 0:** Examine patch topology and patch parameters
  - Download, compile, and run project *TessViewer* from IS
  - Try different parameters
  - No code to write :-)

Task: Tessellate Utah teapot

- Use *quads* to smoothly tessellate 32 Bezier patches, each with 16 control points



Task: Tessellate Utah teapot

- **Task 1:** Tessellate Utah teapot in a very simple way
  - Already done: Vertex shader transforms the positions of control points into world space.
  - Already done: Tessellation control shader passes the data from input to output, and sets tessellation factors to a constant value.
  - **Task**: In *teapot_tess_eval.glsl*, compute the position of vertex in world space, transform it with the view and projection matrices, and store it into *gl_Position*. Also, send the untransformed one (in world space) to fragment shader.
  - Already done: Fragment shader outputs simple white color.
  - Use wireframe to see the result.

Tessellating Bezier patch

- 1D cubic Bezier curve:

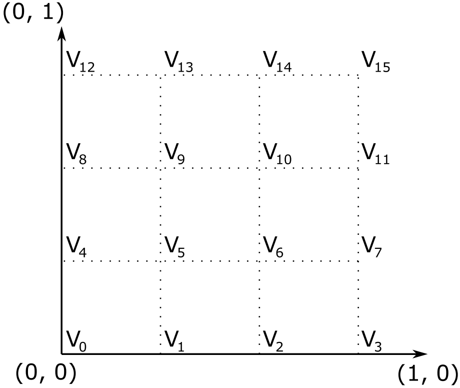
$$bezier4(V_0, V_1, V_2, V_3, t) =$$
$$V_0(1-t)^3 + 3V_1(1-t)^2 t + 3V_2(1-t)t^2 + V_3 t^3$$

- 2D cubic Bezier patch:

$$bezier4x4(V_0 \ldots V_{15}, t_x, t_y) =$$
$$r_0 = bezier4(V_0, V_1, V_2, V_3, t_x)$$
$$\ldots$$
$$r_3 = bezier4(V_{12}, V_{13}, V_{14}, V_{15}, t_x)$$
$$result = bezier4(r_0, r_1, r_2, r_3, t_y)$$

Tessellating Bezier patch

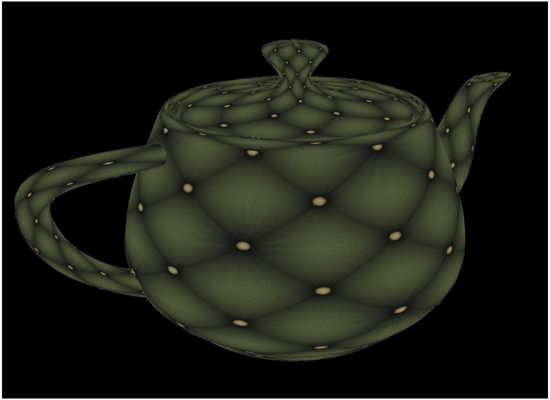
- Order of control points in out patches:



Task: Add texture coordinates

- **Task 2:** Add texture coordinates and texturing
  - Use the *gl_TessCoord* as the texture coordinate, send it from TES to FS.
  - In *teapot_tess_fragment.glsl*, use the texture coordinate to sample the color from *color_tex*.
  - We still do not compute the lighting.

Task: Add texture coordinates

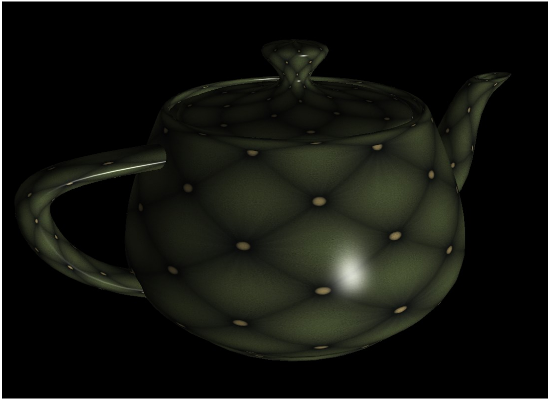


Result

Task: Add lighting

- **Task 3:** Compute the normal and lighting
  - Tangent: direction on the surface of *tex_coord.x* axis
  - Bitangent: direction on the surface of *tex_coord.y* axis
  - Both are precomputed at control points
  - Both are also already transformed into world space in vertex shader
  - And are already passed through TCS to TES
  - **Task**: In TES, evaluate them the same way as positions. Also compute the normal as the cross product between them (order is $\vec{n} = \vec{t} \times \vec{b}$. Pass all three vectors to FS.
  - In fragment shader, use the normal to compute the lighting.
- Optional homework: Compute the tangent and bitangent as derivation of the position.

Task: Add lighting



Result

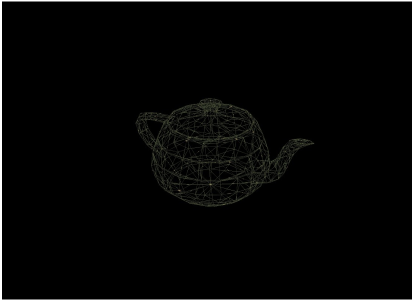
Adaptive tessellation

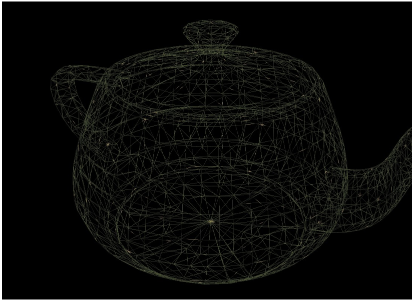
- One of many places where tessellation shaders can be used
- Use more triangles when/where necessary
  - when the object is close
  - where there are more geometry details
  - at the countours
  - discard the patch when outside of the view
  - ...

Task: Add adaptive tessellation

- **Task 4:** Change TCS, use more triangles when the object is closer to the viewer
  - Transform each control point with projection and view matrices, divide it with *w*, and multiply it with the window size to get its position on the screen in pixels. Use barrier to wait for all vertices to be computed.
  - In zeroth invocation, compute the approximation of the length of the four sides of the patch, e.g. sum the length of the three sublines.
  - Divide these lengths with *triangle_size* to get the number of triangles to be tesselated. Use these values as *gl_TessLevelOuter*.
  - Average the two opposites outer levels to get *gl_TessLevelInner*.

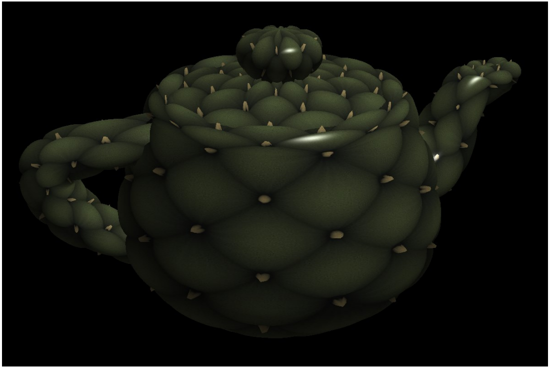
Task: Add adaptive tessellation





Result

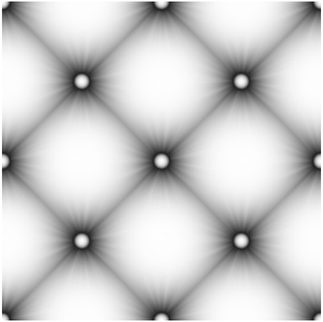
Displacement mapping



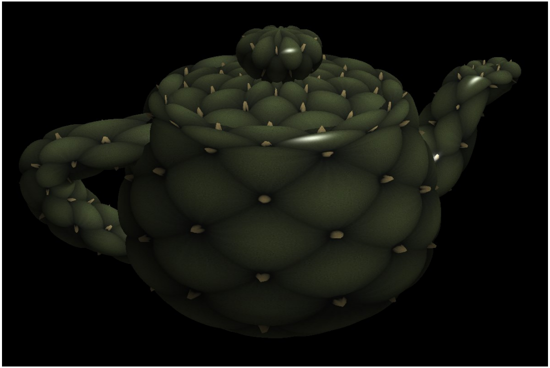
Displacement mapping

Task: Add displacement mapping

- **Task 5:** Displace vertices to add some more geometric details
  - In TES, sample the *height_tex* texture (use function *textureLod*)
  - Multiply the value with *max_displacement* and *height_scale*.
  - Move the position in the direction of the normal.

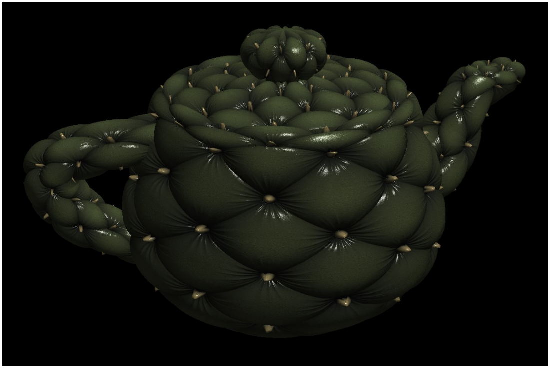


Displacement mapping



Result, notice incorrect lighting

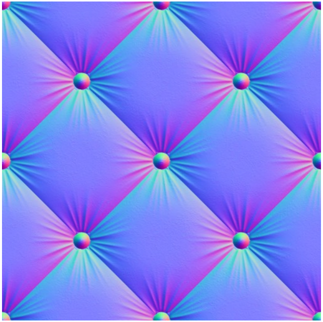
Normal mapping



Normal mapping

Normal mapping

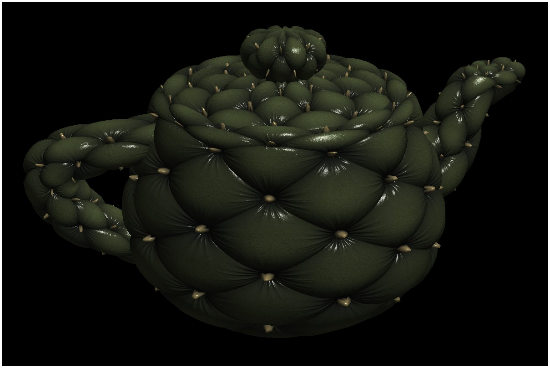
- Get normal from texture, and transform it from (0, 1) to (-1, 1). Don't forget it is in tangent space, i.e. relative to the surface.
  $n^{ts} = texture(tex) \cdot 2 - 1$
- Transform it into world space:
  $n^{ws} = tangent^{ws}_{tes} \cdot n^{ts}_{.x} + bitangent^{ws}_{tes} \cdot n^{ts}_{.y} + normal^{ws}_{tes} \cdot n^{ts}_{.z}$
- Use this normal to compute the lighting



Task: Add normal mapping

- **Task 6:** Implement normal mapping in fragment shader

Task: Add normal mapping



Result