Hardware pro počítačovou grafiku
NPGR019

Geometry & tessellation shaders

Jan Horáček

http://cgg.mff.cuni.cz/
MFF UK Praha

2012

Obsah

Propojení vertex shaderu s geometrickými daty

- vertexová data (pozice, tex. koordináty, normály. . .) vstupují do OpenGL pipeline přes **vertex shader**
- je nutné spojit **zdroj** vertexových dat s **proměnnou** v shaderu
  - proto je potřeba znát pozici atributu
- pozice atributu může být
  - explicitně definovaná použitím kvalifikátoru `layout`
  - explicitně definovaná funkcí `glBindAttribLocation()`
  - zjištěna automatická pozice voláním `glGetAttribLocation()`

Propojení vertex shaderu s geometrickými daty

- data v oddělených částech VBO

data jednoho typu spojitě jako pole

```
#define BUFFER_OFFSET(addr)   ((GLvoid*) (addr))

GLsizeiptr offset = 0;
glVertexAttribPointer( vPos, 3, GL_FLOAT,
      GL_FALSE, 0, BUFFER_OFFSET(offset) );

offset += sizeof(v);
glVertexAttribPointer( vColor, 4, GL_UNSIGNED_BYTE,
      GL_TRUE, 0, BUFFER_OFFSET(offset) );

offset += sizeof(c);
glVertexAttribPointer( vTexCoord, 2, GL_FLOAT,
      GL_FALSE, 0, BUFFER_OFFSET(offset) );
```

Propojení vertex shaderu s geometrickými daty

- data prokládána ve VBO

data jako struktury ve VBO

```
struct VertexData
{
    GLfloat tc[2];
    GLubyte  c[4];
    GLfloat  v[3];
}
```

```
GLsizei stride = sizeof( VertexData );
GLsizeiptr offset = 0;

glVertexAttribPointer( vTexCoord, 2, GL_FLOAT,
      GL_FALSE, stride, BUFFER_OFFSET(offset) );
offset += sizeof(v.tc);

glVertexAttribPointer( vColor, 4, GL_UNSIGNED_BYTE,
      GL_TRUE, stride, BUFFER_OFFSET(offset) );
offset += sizeof(v.c);

glVertexAttribPointer( vPos, 3, GL_FLOAT,
      GL_FALSE, stride, BUFFER_OFFSET(offset) );
```

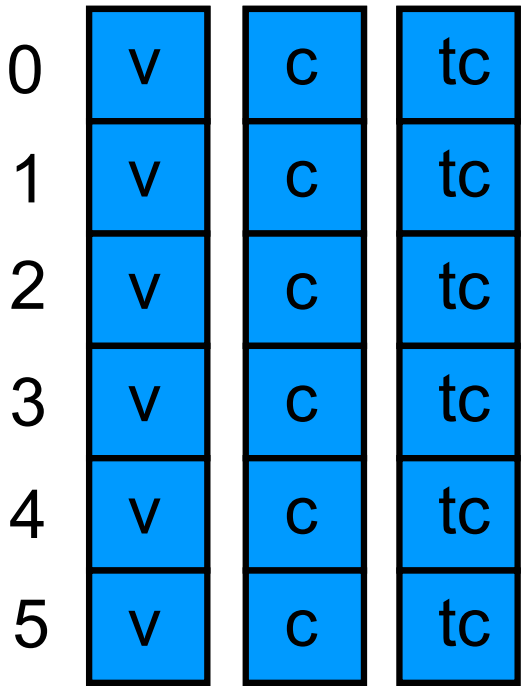
Propojení vertex shaderu s geometrickými daty

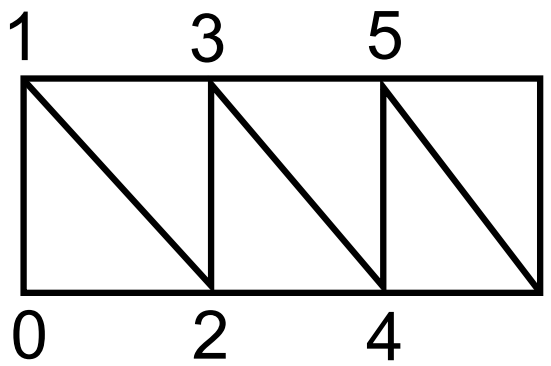
- je potřeba oznámit OpenGL, která vertexová pole budou použita
  - `glEnableVertexAttribArray( vPos );`
  - `glEnableVertexAttribArray( vColor );`
  - `glEnableVertexAttribArray( vTexCoord );`

Vykreslování geometrických primitiv

- pro spojité pole vertexů (po částech i prokládané)

```
glDrawArrays( GL_TRIANGLE_STRIP, 0, n );
```

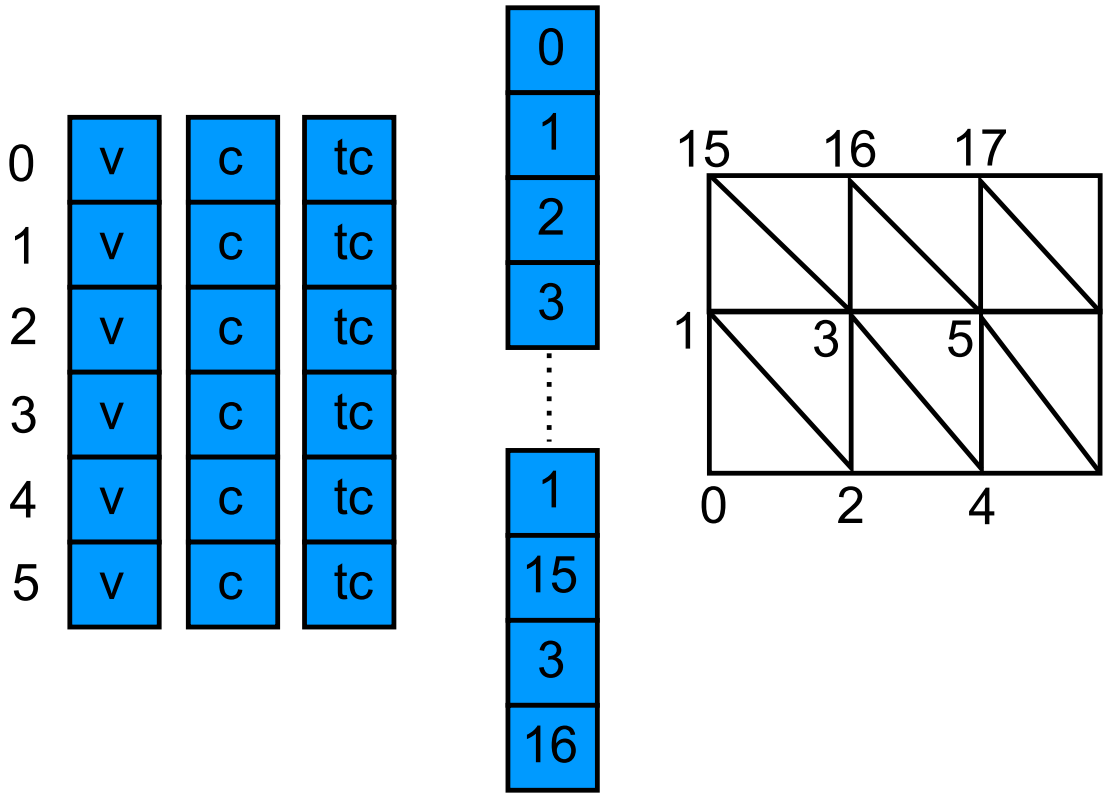




Vykreslování geometrických primitiv

- pro indexované skupiny vertexů

```
glDrawElements( GL_TRIANGLE_STRIP, n,
        GL_UNSIGNED_SHORT, BUFFER_OFFSET(offset) );
```



Instancované vykreslování

- posílání vícekrát stejné geometrie do OpenGL
- každá iterace je nová **instance**
- vertex shader může měnit vykreslování v závislosti na tom, která instance se zrovna vykresluje
- číslo iterace uloženo v GLSL proměnné `gl_InstanceID`

```
glDrawArraysInstanced( GL_TRIANGLE_STRIP,
      0, n, 10 );

glDrawElementsInstanced( GL_TRIANGLE_STRIP, n,
      GL_UNSIGNED_INT, BUFFER_OFFSET(0), 10 );
```

Variace na glDrawElements()

- občas se hodí přidat offset k seznamu vykreslovanému přes `glDrawElements()`
  - toto umožňuje použít stejnou topologii pro různou geometrii
- všechny `glDrawElements()` varianty mají i verzi pro offset
  - `glDrawElementsBaseVertex()`
  - `glDrawRangeElementsBaseVertex()`
  - `glDrawElementsInstancedBaseVertex()`
- stejné parametry, pouze je zde navíc jeden parametr určující offset
  - `glDrawElementsBaseVertex( GL_TRIANGLES, 0 GL_UNSIGNED_INT, BUFFER_OFFSET(offset), 15 );`

Nepřímé vykreslování

- parametry pro volání `glDrawArraysInstanced()` se dají uložit do `GL_DRAW_INDIRECT_BUFFER`

data musí mít pevnou strukturu

```
struct DrawArraysIndirectCommand
{
   GLuint count;
   GLuint primCount;
   GLuint first;
   GLuint reservedMustBeZero;
};

// volání
glDrawArraysIndirect( primType, BUFFER_OFFSET(offset) );
```

- toto je připraveno pro budoucí způsoby vykreslování
  - například dynamicky vytvářená geometrie

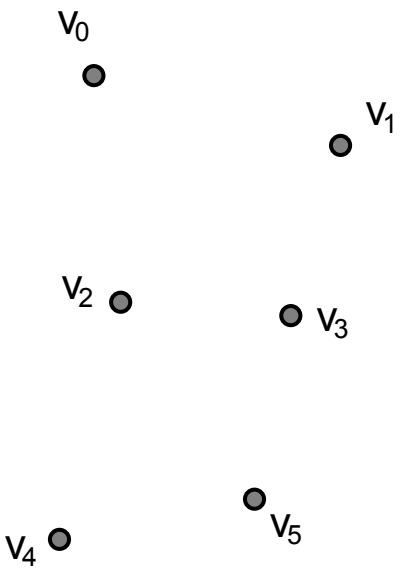
Nepřímé vykreslování

- podobně existuje i verze pro indexované vykreslování `glDrawElementsIndirect()`

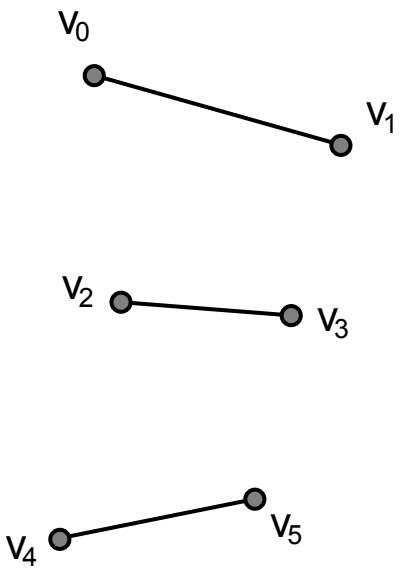
struktura pro glDrawElementsIndirect()

```
struct DrawElementsIndirectCommand
{
   GLuint count;
   GLuint primCount;
   GLuint firstIndex;
   GLint baseVertex;
   GLuint reservedMustBeZero;
};
```

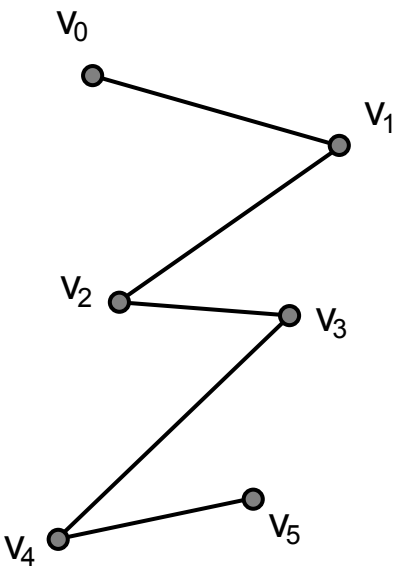
Geometrická primitiva I



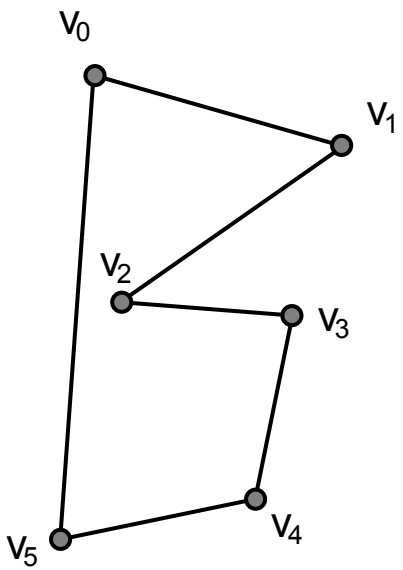
GL_POINTS



GL_LINES

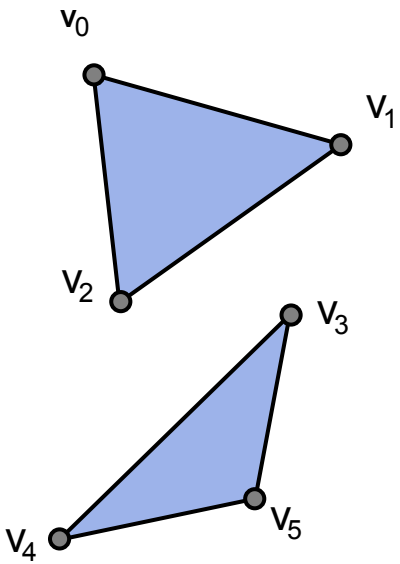


GL_LINE_STRIP

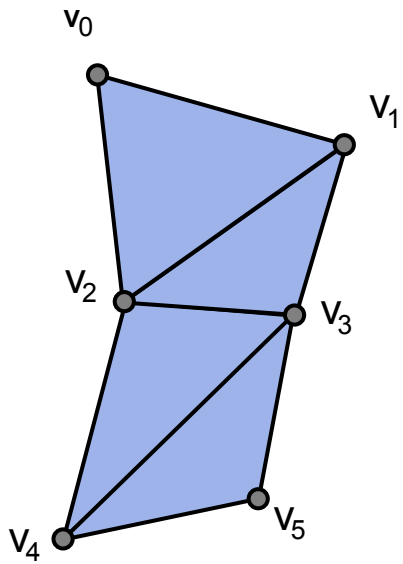


GL_LINE_LOOP

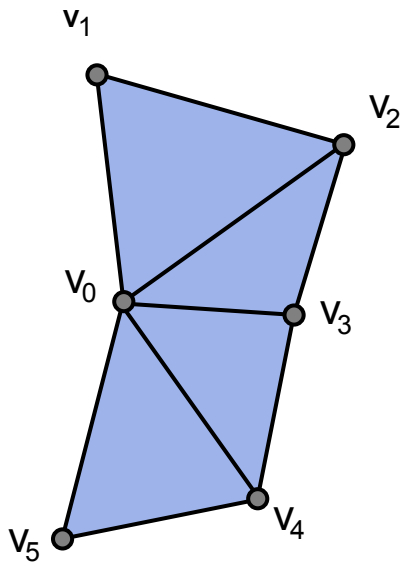
Geometrická primitiva II



GL_TRIANGLES

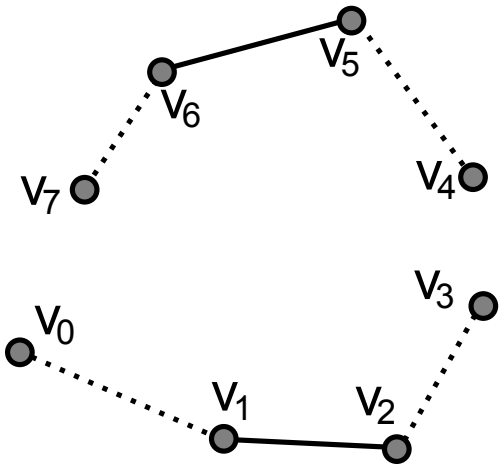


GL_TRIANGLE_STRIP

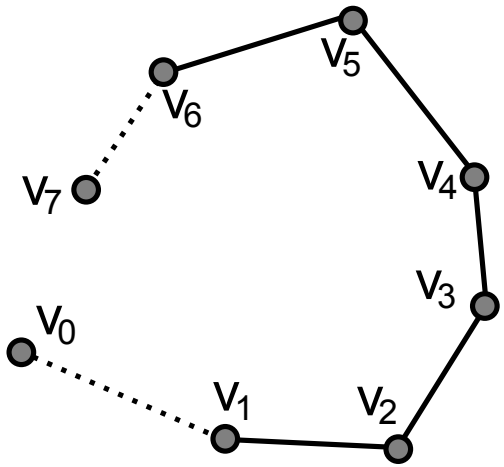


GL_TRIANGLE_FAN

Geometrická primitiva III

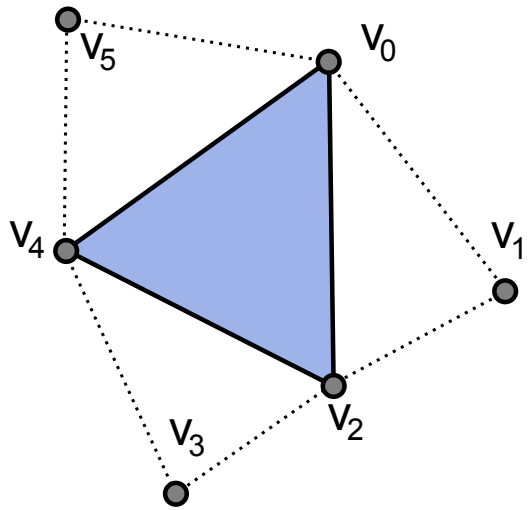


GL_LINES_ADJACENCY

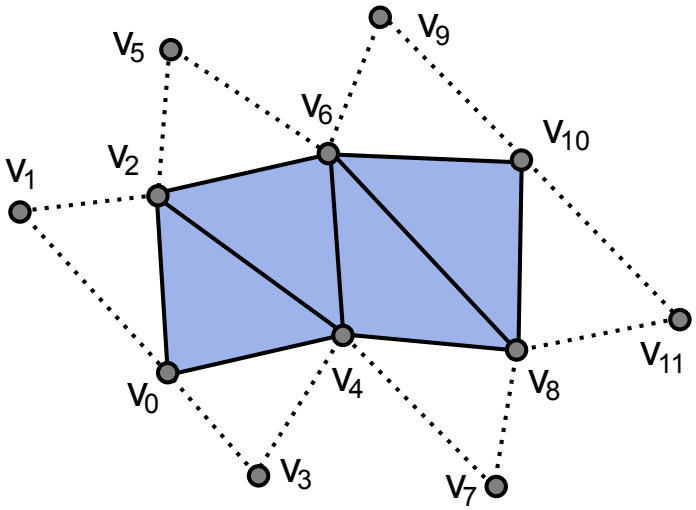


GL_LINE_STRIP_ADJACENCY

Geometrická primitiva IV

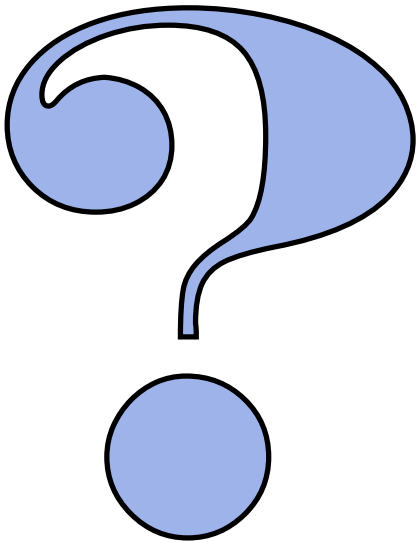


GL_TRIANGLES_ADJACENCY



GL_TRIANGLE_STRIP_ADJACENCY

Geometrická primitiva V



GL_PATCHES

Obsah

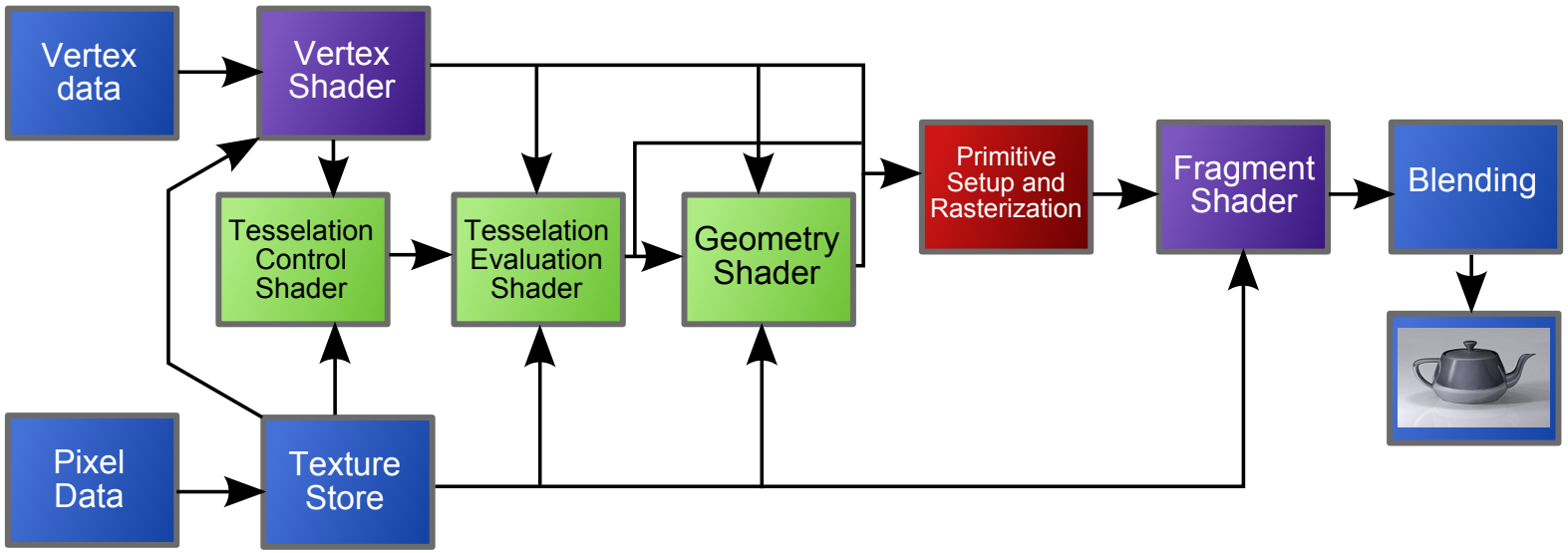
Detaily scény

- teselace se dá považovat za další krok v **přidávání detailů** do scény
- historicky:
  1. **jednobarevné** a **gouraudově stínované** polygony - holé objekty
  2. **textury** - barevný detail
  3. **bumpmapping** (ve všech formách od emboss-bumpmappingu až po normalmapping/procedurální generování normál) - detail pro stínování, obzvláště dynamická světla
  4. **parallax mapping** - *falešný* geometrický detail
  5. **teselace** - *skutečný* geometrický detail
- toto je ovšem pouze **jedno** z možných využití teselace

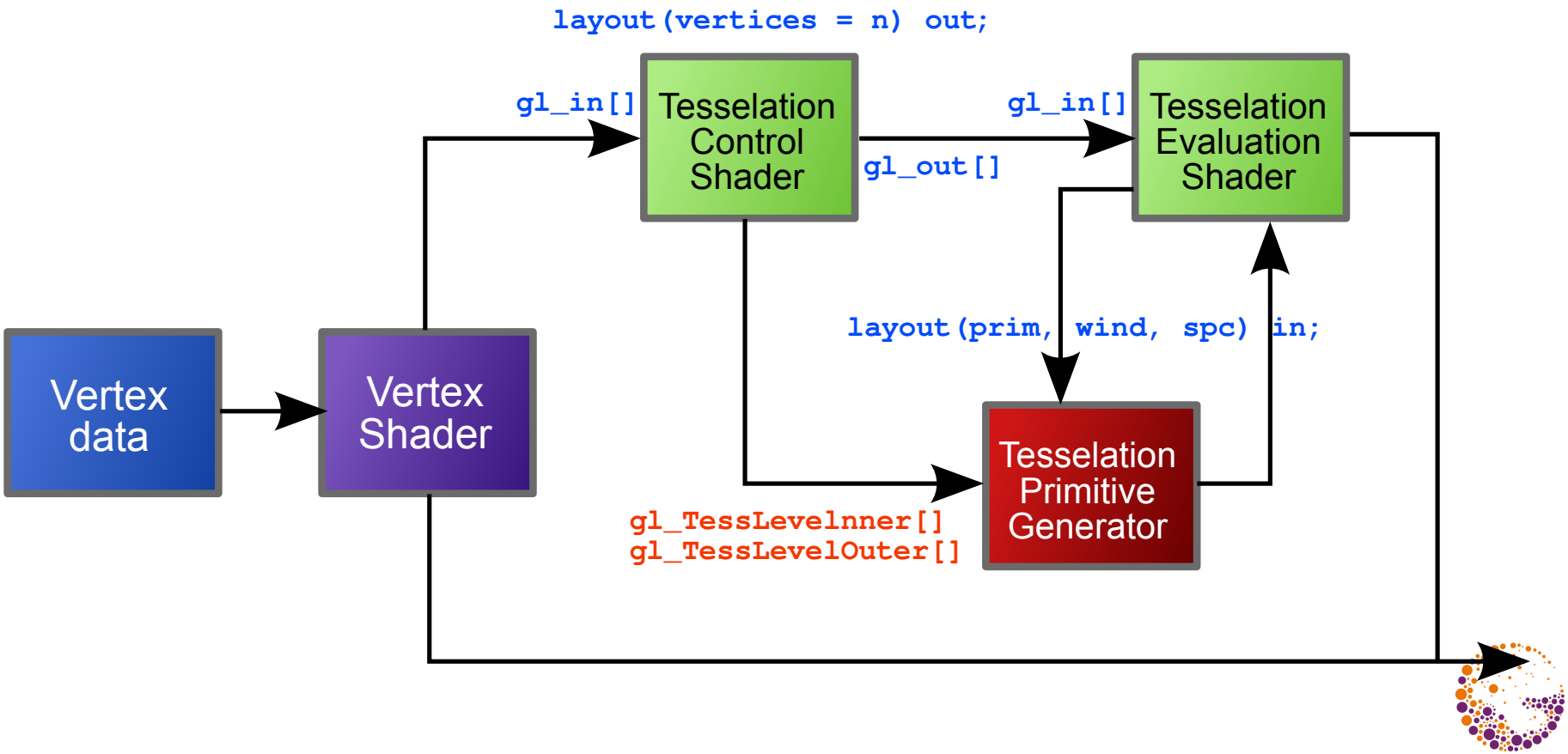
Přehled

- teselace se využívá ke **generování** geometrie
- pouze jeden typ primitiv - **patches**
  - do `glDraw*()` se uvede konstanta `GL_PATCHES`
- 3 fáze: 1 konfigurovatelná, 2 programovatelné pomocí shaderů
- **tessellation control shader**
  - zpracovává sadu vstupních vertexových atributů
  - generuje výstupní *patch*ové atributy
  - definuje teselační parametry
- **tessellation evaluation shader**
  - zpracovává výstupní *patch*ové atributy
  - počítá finální vertexy generovaných primitiv

OpenGL pipeline



Tok dat v teselační části pipeline



Tok dat v teselační části pipeline

- teselační shadery pracují na sadách vrcholů
  - vstupní i výstupní data jsou v polích
- proměnná `gl_in[]` obsahuje vstupní data pro oba shadery
  - délka pole = `gl_in.length()`
- proměnná `gl_out[]` má přepočítané informace o vrcholech

obě jsou pole struktur

```
in gl_PerVertex {
    vec4  gl_Position;
    float gl_PointSize;
    vec4  gl_ClipDistance[];
} gl_in[];
```

Tesselation control shader

- výstupní layout `vertices` specifikuje počet výstupních vertexů
- control shader spuštěn pro každý výstupní vertex
- vstupem i výstupem `patch`
- má přístup ke všem vstupním atributům
- smí zapisovat jen do *svého* výstupu
  - podle `gl_InvocationID`
- základní úkol je připravit parametry pro teselaci a přepočítat `patch`

Příklad tessellation control shaderu

tessellation control shader

```
#version 400 core
layout (vertices = 4) out;

uniform float Inner;
uniform float Outer;

void main()
{
   gl_TessLevelInner[0] = Inner;
   gl_TessLevelInner[1] = Inner;
   gl_TessLevelOuter[0] = Outer;
   gl_TessLevelOuter[1] = Outer;
   gl_TessLevelOuter[2] = Outer;
   gl_TessLevelOuter[3] = Outer;

   gl_out[gl_InvocationID].gl_Position =
      gl_in[gl_InvocationID].gl_Position;
}
```

Tessellation control bez shaderu

- mnoho programů pouze přeposílá data
- v případě, že vstupní a výstupní *patch* mají stejný počet vertexů, může OpenGL tuto stage nahradit
  1. specifikuj počet vertexů ve vstupní *patch*

     ```
     glPatchParameteri( GL_PATCH_VERTICES, NumVertices );
     ```

  2. specifikuj vnitřní a vnější teselační parametry

     ```
     GLfloat outer[4], inner[2];
     glPatchParameterfv( GL_PATCH_DEFAULT_OUTER_LEVEL,
                         outer );
     glPatchParameterfv( GL_PATCH_DEFAULT_INNER_LEVEL,
                         inner );
     ```

Generování primitiv

- teselace vytváří geometrická primitiva rozdělením parametrického prostoru
- k dispozici jsou tři typy parametrizací

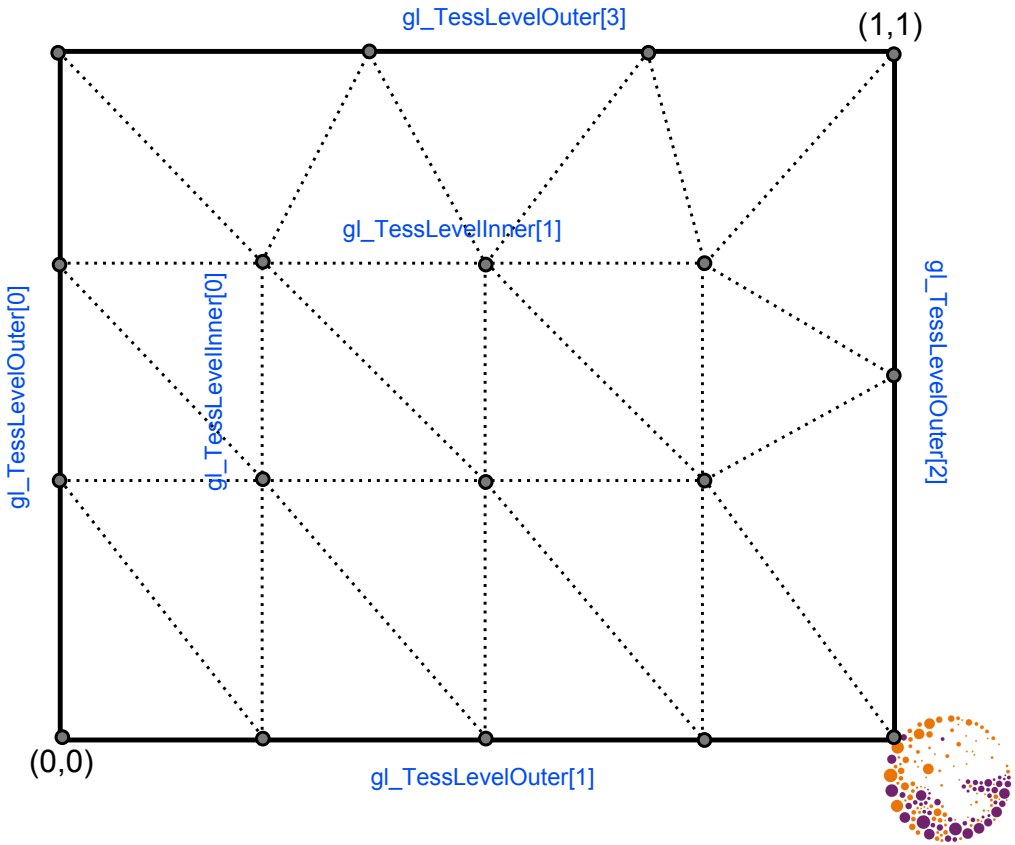
Typ	Parametrický prostor	Teselační faktory
quad	jednotkový čtverec: (u, v) u, v $\in$ [0, 1]	`gl_TessLevelInner`: 0 ... 1 `gl_TessLevelOuter`: 0 ... 3
triangle	barycentrické: (u, v, w) u, v, w $\in$ [0, 1] u + v + w = 1	`gl_TessLevelInner`: 0 `gl_TessLevelOuter`: 0 ... 2
isolines	čára: (u, v) u, v $\in$ [0, 1] u se na čáře mění, v je konstantní	`gl_TessLevelOuter`: 0 ... 1

Příklad teselace čtyřúhelníku

parametry

```
gl_TessLevelInner[0] = 3.0;
gl_TessLevelInner[1] = 4.0;
gl_TessLevelOuter[0] = 3.0;
gl_TessLevelOuter[1] = 4.0;
gl_TessLevelOuter[2] = 2.0;
gl_TessLevelOuter[3] = 3.0;

(using equal_spacing)
```

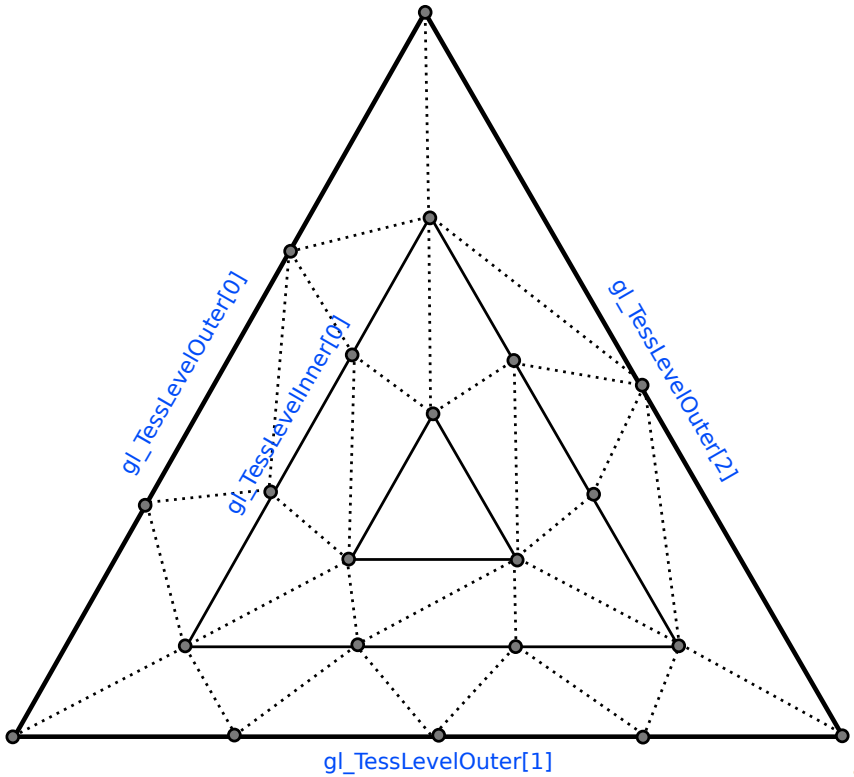


Příklad teselace trojúhelníku

parametry

```
gl_TessLevelInner[0] = 5.0;
gl_TessLevelOuter[0] = 3.0;
gl_TessLevelOuter[1] = 4.0;
gl_TessLevelOuter[2] = 2.0;

(using equal_spacing)
```

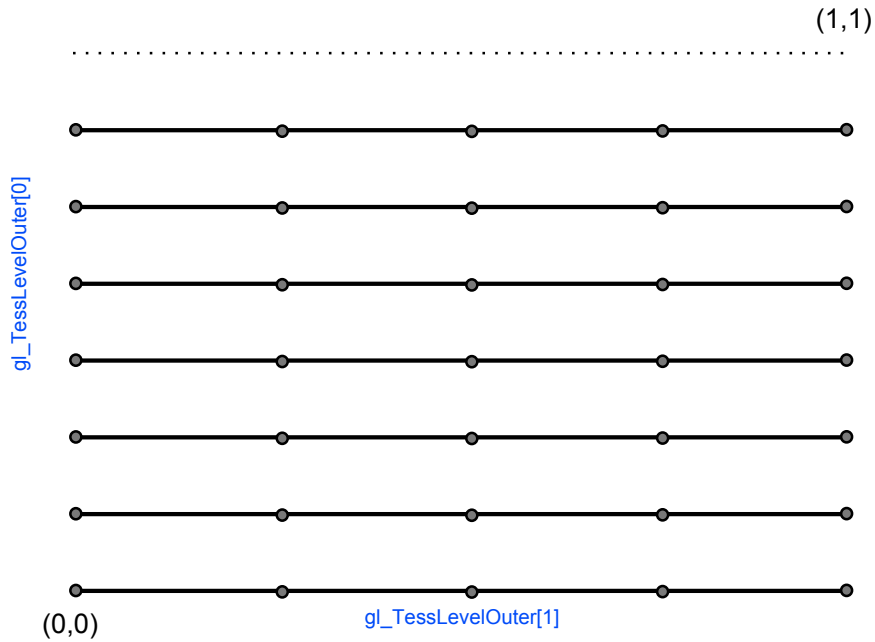


Příklad teselace čar

parametry

```
gl_TessLevelOuter[0] = 7.0;
gl_TessLevelOuter[1] = 4.0;

(using equal_spacing)
```



Příklad tessellation evaluation shaderu

tessellation evaluation shader

```
#version 400 core
layout (quads, equal_spacing, ccw) in;
uniform mat4 MV, P;

float B( int i, float u) {
   const vec4 bc = vec4( 1, 3, 3, 1 );
   return bc[i] * pow( u, i ) * pow( 1.0 - u, 3 - i );
}

void main() {
   float u = gl_TessCoord.x, v = gl_TessCoord.y;

   vec4 pos = vec4( 0.0 );
   for( int j = 0; j < 4; j++ )
      for( int i = 0; i < 4; i++ )
         pos += B( i, u ) * B( j, v ) * gl_in[4*j+i].gl_Position;
   gl_Position = P * MV * pos;
}
```

Kontrolování pozic teselace

- teselační faktory jsou čísla s plovoucí řád. čárkou
- různé módy kontrolují, jak se bude strana rozdělovat
- každá hrana se může rozdělit na max. `GL_MAX_TESS_GEN_LEVELS`, aktuálně to je 64

Mód teselace	**Interval hodnot**
`equal_spacing`	[1, *max*]
`fractional_even_spacing`	[2, *max*]
`fractional_odd_spacing`	[1, *max-1*]

- `equal_spacing` akceptuje celá čísla (zaokrouhluje nahoru), vytvoří *n* stejných intervalů
- `fractional_...` teselace zaokrouhlí na nejbližší vyšší sudé/liché číslo, *(n-2)* intervalů je stejných a 2 většinou menší intervaly jsou závislé na desetinné části hodnoty

Primitive winding a Point mode

- vytvořená primitiva mají vrcholy implicitně **proti** směru hodinových ručiček (`ccw`)
  - použijte `cw` pro pořadí **po** směru hodinových ručiček
- místo polygonů (trojůhelníků) se dají generovat **body**, stačí nastavit `point_mode` v direktivě `layout`

příklad nastavení layoutu v evaluation shaderu

```
layout( triangles, cw, fractional_even_spacing,
        point_mode ) in;
```

Obsah

Přehled

- poslední *volitelná* shader stage před rasterizerem
- svým způsobem podobné teselační části
  - generování nové geometrie
  - dosahuje **podobných** výsledků za pomocí **jiných** principů
  - programátor specifikuje počet a typ vygenerovaných primitiv

Vstup

- vstupem *assembled* primitives (tedy ne *strips* nebo *fan*)
  - `points` ...1
  - `lines` ...2
  - `lines_adjacency` ...4
  - `triangles` ...3
  - `triangles_adjacency` ...6
- má veškerou informaci o **celém** primitivu, nejen o 1 vrcholu
- nová primitiva se **sousedností** (*adjacency*) dodávají informaci o bezprostředním okolí

Vestavěné proměnné

```
in gl_PerVertex {
    vec4  gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_in[];

in int gl_PrimitiveIdIn;

// pouze v OpenGL 4.0+
in int gl_InvocationID;
```

Výstup

- možná výstupní primitiva:
  - `points`
  - `line_strip`
  - `triangle_strip`
- typy vstupního a výstupního primitiva jsou **nezávislé**
  - vstup je po vykonání shaderu **zapomenut**
- výstupem **0** nebo **více** primitiv
  - např. dva `triangle_strips`, každý ze 3 trojůhelníků
- *pozn.*:vzhledem k původní myšlence geometry shaderů **nemusí** být implementace nutně optimalizovaná pro **masivní** tvorbu geometrie
  - na to je teselační část pipeline

Výstupní proměnné

Vestavěné proměnné

```
vec4  gl_Position;
float gl_PointSize;
float gl_ClipDistance[];

out int gl_PrimitiveID;
out int gl_Layer;

// pouze v OpenGL 4.0+
out int gl_ViewportIndex;
```

Příklad geometry shaderu

geometry shader

```
#version 400 core
layout (triangles, invocations = 1) in;
layout (triangle_strip, max_vertices = 3) out;
uniform float scale;

void main() {
   vec4 v[3], center = vec4(0);
   for( int i = 0; i < 3; i++ ) {
      v[i] = gl_in[i].gl_Position;
      center += v[i];
   }
   center /= 3;
   for( int i = 0; i < 3; i++ ) {
      gl_Position = mix( v[i], center, scale );
      EmitVertex();
   }
   EndPrimitive();
}
```

Geometry vs. Tessellation

Problém	**Geometry**	**Tessellation**
generování primitiv	**explicitní kontrola**: specifikuje se pozice i propojení	**parametrická kontrola**: teselační parametry určují počet vygenerovaných primitiv
topologie	**velmi lokalizovaná**: je vidět jen nejbližší okolí s limitovanou konektivitou	**implicitně propojené**: všechna primitiva jsou propojená, upravuje se jen umístění vrcholu
zdrojová primitiva	**limitovaná sada**: v podstatě jen základní primitiva, body, čáry, trojůhelníky	**jakákoliv patch**: OpenGL vidí jen seznam vrcholů, vztahy se definují explicitně
praskliny mezi primitivy	**problematické**: je třeba dávat velký pozor na umístění vrcholů	**jednodušší**: spoj je *vodotěsný*, pokud jsou teselační parametry stejné, praskání limitováno jen na výpočetní přesnost

Obsah

Literatura

- OpenGL Architecture Review Board: **OpenGL Programming Guide: The Official Guide to Learning OpenGL**, Addison-Wesley, nejnovější vydání (aktuálně 8. vydání pro OpenGL 4.1)
- Randi J. Rost, Bill Licea-Kane: **OpenGL Shading Language, 3rd Edition**, Addison-Wesley
- The Khronos Group: **The OpenGL Graphics System: A Specification (Core/Compatibility profile)**, `http://www.opengl.org/registry/`
- Christophe Riccino: **OpenGL reviews**, `http://www.g-truc.net/post-opengl-review.html`
- Wikipedia: `http://en.wikipedia.org/wiki/OpenGL`
- OpenGL tutorials: `http://nehe.gamedev.net`