matfyz

Computer Graphics Charles University

Geometry and tessellation on GPU

© 2012-2016 Josef Pelikán, Jan Horáček
CGG MFF UK Praha

pepca@cgg.mff.cuni.cz
http://cgg.mff.cuni.cz/~pepca/

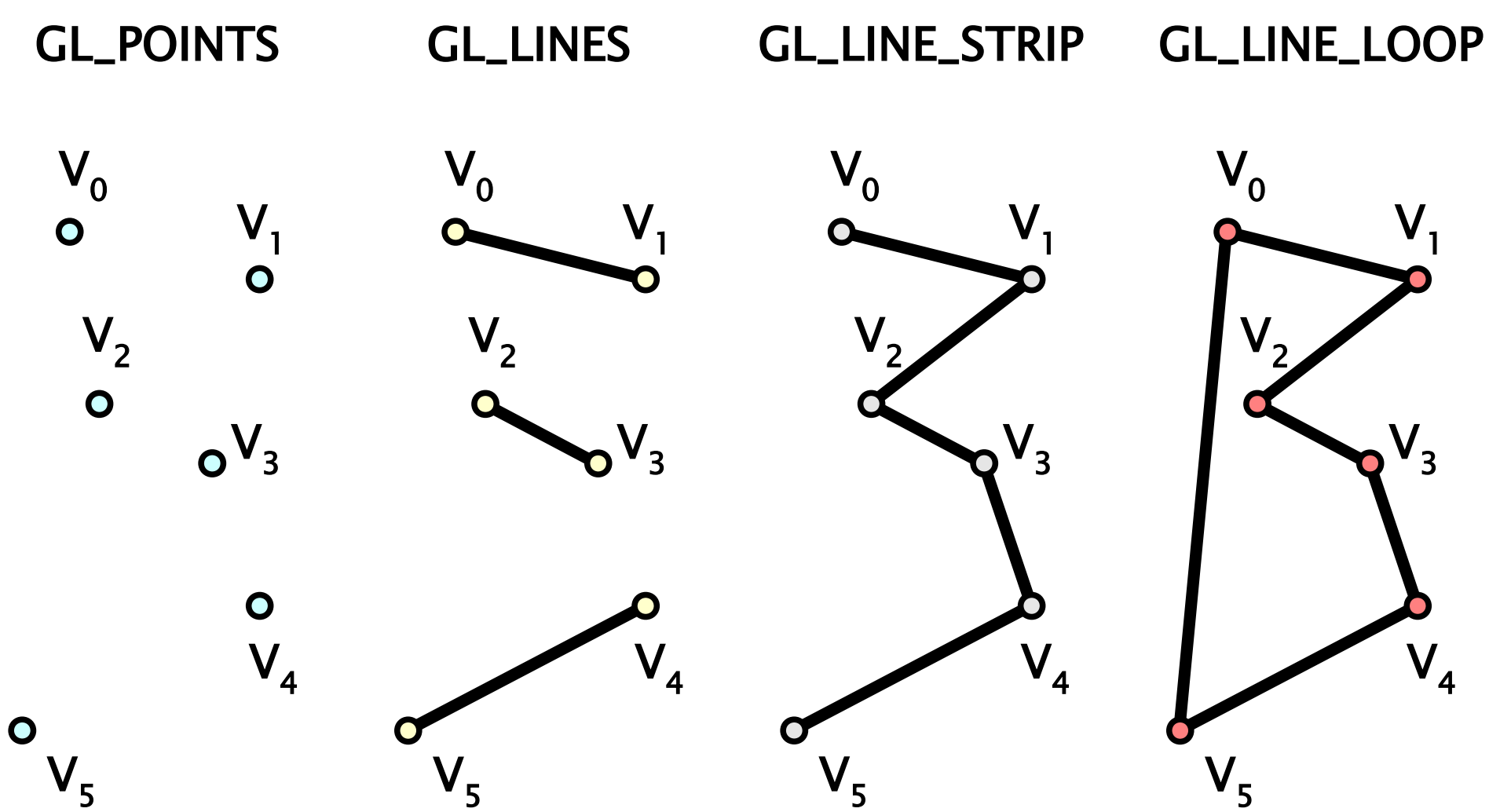
Content

- advanced drawing modes in modern OpenGL
  - primitive types
  - vertex attributes
  - indirect drawing
  - instanced drawing, index offsets
- **Tessellation shaders**
  - tessellation control shader
  - tessellation evaluation shader
- **Geometry shaders**

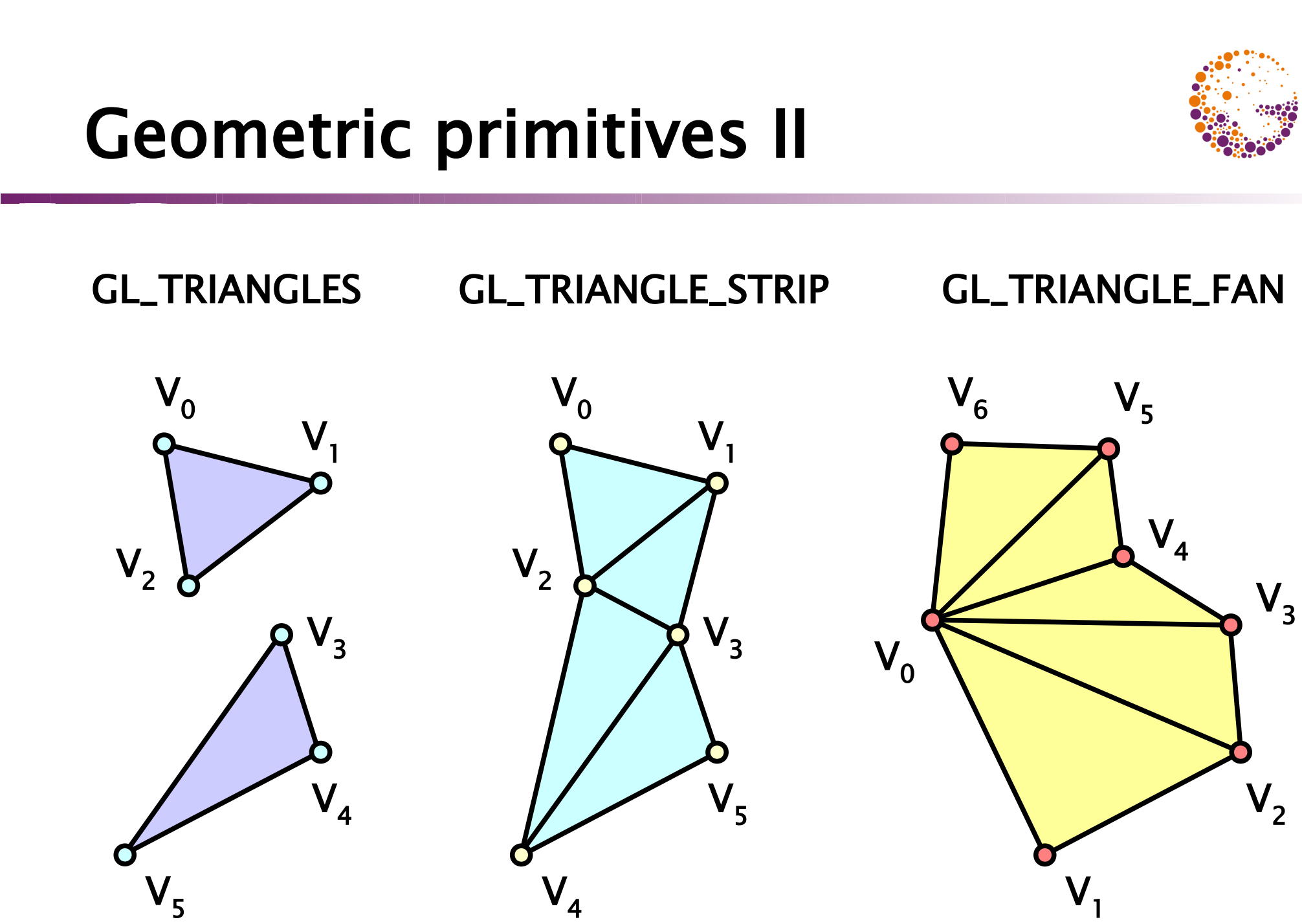
Geometry primitives

- **deprecated primitives**
  - `GL_QUADS, GL_QUAD_STRIP`
  - `GL_POLYGON`
- **new primitives with adjacency** (for subdivision..)
  - **`GL_LINES_ADJACENCY`**
  - **`GL_LINE_STRIP_ADJACENCY`**
  - **`GL_TRIANGLES_ADJACENCY`**
  - **`GL_TRIANGLE_STRIP_ADJACENCY`**

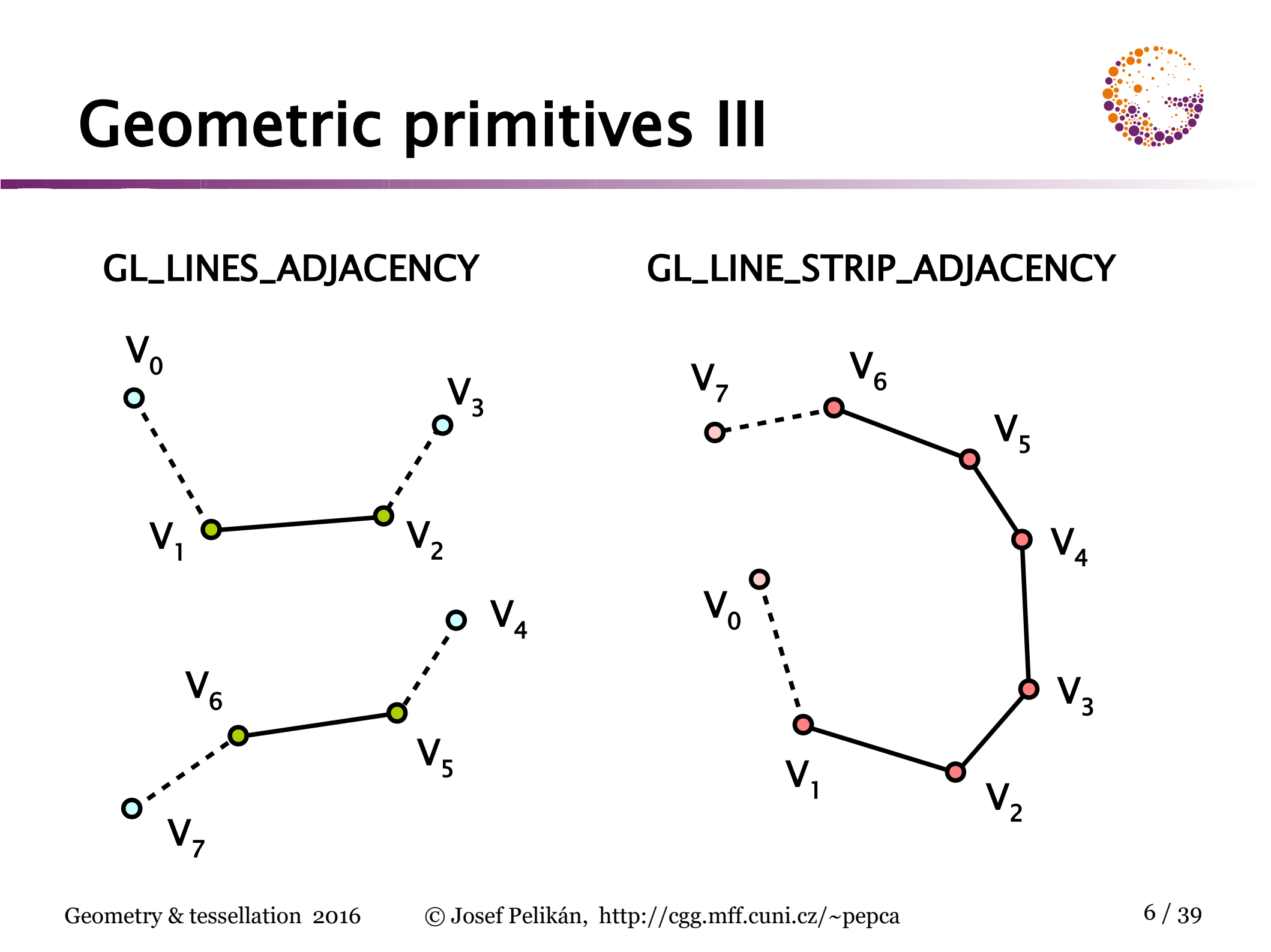
Geometric primitives I



Geometric primitives II



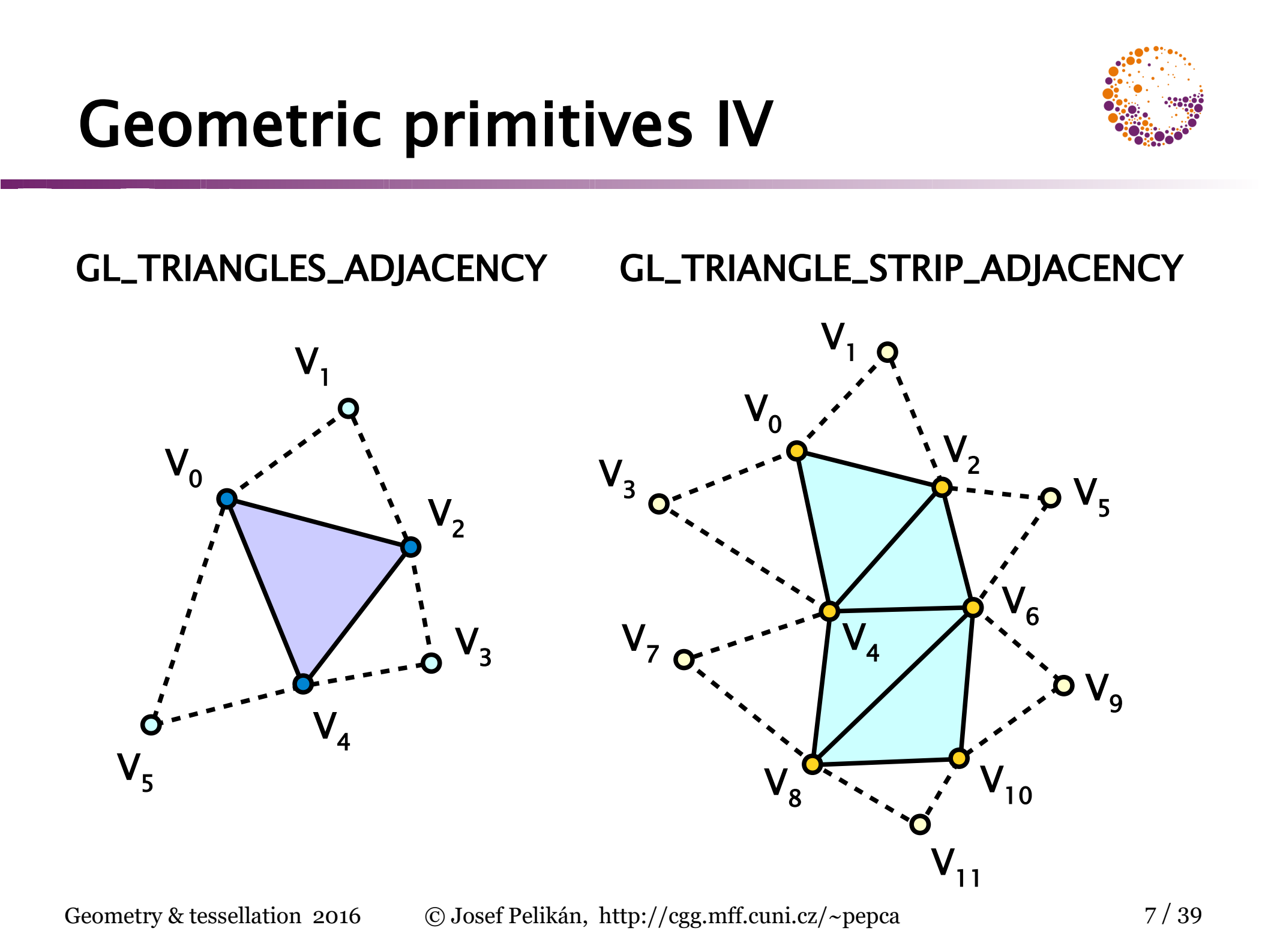
Geometric primitives III



GL_LINES_ADJACENCY

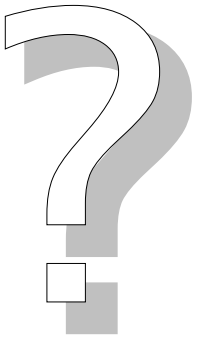
GL_LINE_STRIP_ADJACENCY

Geometric primitives IV



Geometric primitives V

GL_PATCHES



$V_0 \quad \cdots \quad V_{N-1}$

Vertex data

- **vertex attributes**
  - need to be bound to shader variables
- attribute position definition
  - explicit – `layout` (in shader)
  - explicit – `glBindAttribLocation()`
  - automatic – `glGetAttribLocation()`

Interlaced example

```
struct VertexData
{
  GLfloat tc[2];   // texture coordinates
  GLubyte  c[4];   // vertex color
  GLfloat  v[3];   // vertex coordinate
}
Glsizei stride = sizeof( VertexData ); // vertex stride
Glsizeiptr offset = 0;

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

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

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

Interlaced example

- declare which vertex arrays will be used

```
glEnableVertexAttribArray( vPos );
glEnableVertexAttribArray( vColor );
glEnableVertexAttribArray( vTextCoord );
```

Rendering primitives (batch)

- continuous array of vertices **without indices**

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

buffer offset

# of vertices

- **indexed rendering** (using index buffer)

```
glDrawElements( GL_TRIANGLE_STRIP, n,
  GL_UNSIGNED_INT, BUFFER_OFFSET(off) );
```

index type

buffer offset

Instanced rendering

- just **repetition of the same rendering data**
  - vertex shader updates its behavior according to builtin internal integer variable `gl_InstanceID`

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

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

# of instances

Relative indexed rendering

- **index offset**
  - the same topology with different geometry
  - remember “Geometry Clipmaps” (Hoppe et al.)

```
glDrawElementsBaseVertex()
glDrawRangeElementsBaseVertex()
glDrawElementsInstancedBaseVertex()
```

```
glDrawElementsBaseVertex( GL_TRIANGLES, n,
  GL_UNSIGNED_INT, BUFFER_OFFSET(0), 200 );
```

index offset

Indirect rendering

- data are stored in a **server-side buffer** (**GL_DRAW_INDIRECT_BUFFER**)
  - alternatives to classical rendering functions, e.g.:

    ```
    glDrawArraysIndirect()
    glMultiDrawArraysIndirect()
    glMultiDrawElementsIndirect()
    ```

```
glDrawElementsIndirect( GL_TRIANGLES,
  GL_FLOAT, BUFFER_OFFSET(0) );
```

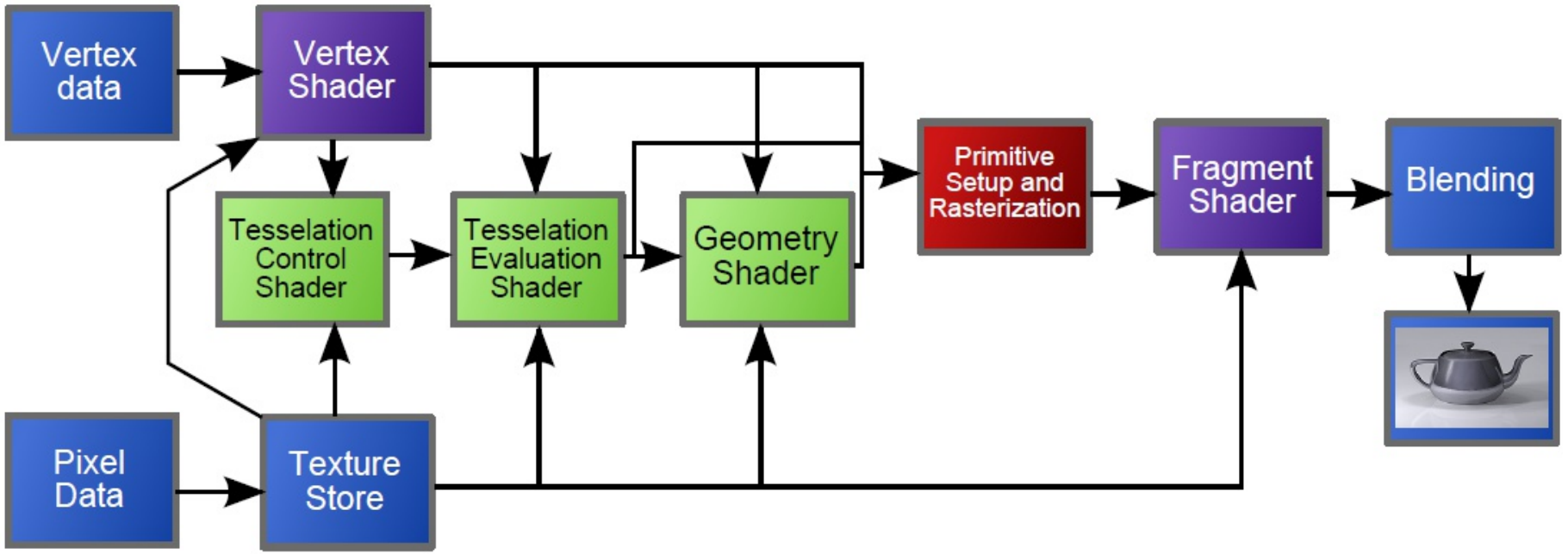
indirect buffer offset

Indirect rendering – buffer data

```
typedef struct
{
  uint count;         // number of vertices (elements)
  uint primCount;     // number of instances
  uint firstIndex;    // index buffer offset
  uint baseVertex;    // vertex offset
  uint baseInstance; // instance offset
} DrawElementsIndirectCommand;
```

Tessellation shaders

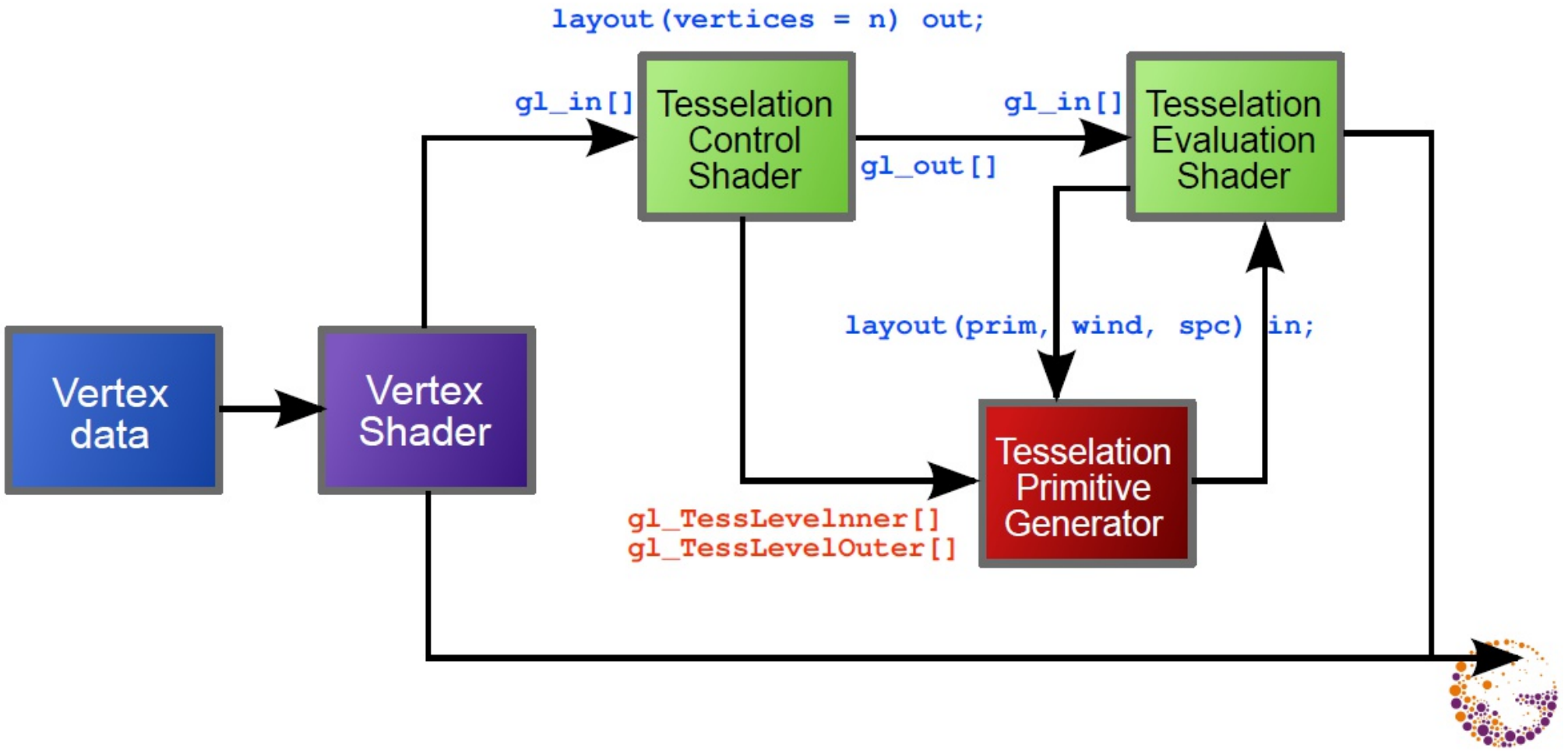
- introduced in OpenGL 4.0 (March 2010)
  - tessellation shaders
    - **tessellation control shader**
    - **tessellation evaluation shader**
  - efficient but geometrically simple surface tesselation



Tessellation shaders

- **geometric primitive – GL_PATCHES**
  - arbitrary semantics – depends on a developer (struct[])
  - 3 modules: one configurable, two programmable ones
- **Tessellation Primitive Generator** (“for” loops)
- **Tessellation Control Shader**
  - in – patch attributes (array of vertex attributes)
  - out – patch attributes (dtto)
  - defines tessellation parameters (for the TPG)
- **Tessellation Evaluation Shader**
  - in – patch attributes (dtto)
  - out – vertex attributes of output primitives

Tessellation modules



Data flow in tessellation subsystem

- tessellation shaders are working on vertex sets
  - input and output data are arrays
- builtin `gl_in[]` variable – input for both tess. shaders
- builtin `gl_out[]` variable (can be modified by TCS)
- **vertex information**: similar to "VertexShader-only" but in an array

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

Tessellation Control Shader

- output layout **`vertices`** defines number of output vertices
- TCS is called for every output patch vertex
- input & output data … patch
  - access to all input attributes
- allowed to write only to its output
  - **`gl_InvocationID`**
- **basic task** of TCS
  - setup tessellation parameters
  - prepare (optionally recompute) output patch data

Tessellation Control Shader example

```
#version 400 core
layout (vertices = 4) out;
uniform float Inner;
uniform float Outer;
void main ()
{
  if ( gl_InvocationID == 0 )
  {
    gl_TessLevelInner[ 0 ] = gl_TessLevelInner[ 1 ] =
      Inner;
    gl_TessLevelOuter[ 0 ] = gl_TessLevelOuter[ 1 ] =
    gl_TessLevelOuter[ 2 ] = gl_TessLevelOuter[ 3 ] =
      Outer;
  }
  gl_out[ gl_InvocationID ].gl_Position =
   gl_in[ gl_InvocationID ].gl_Position;
}
```

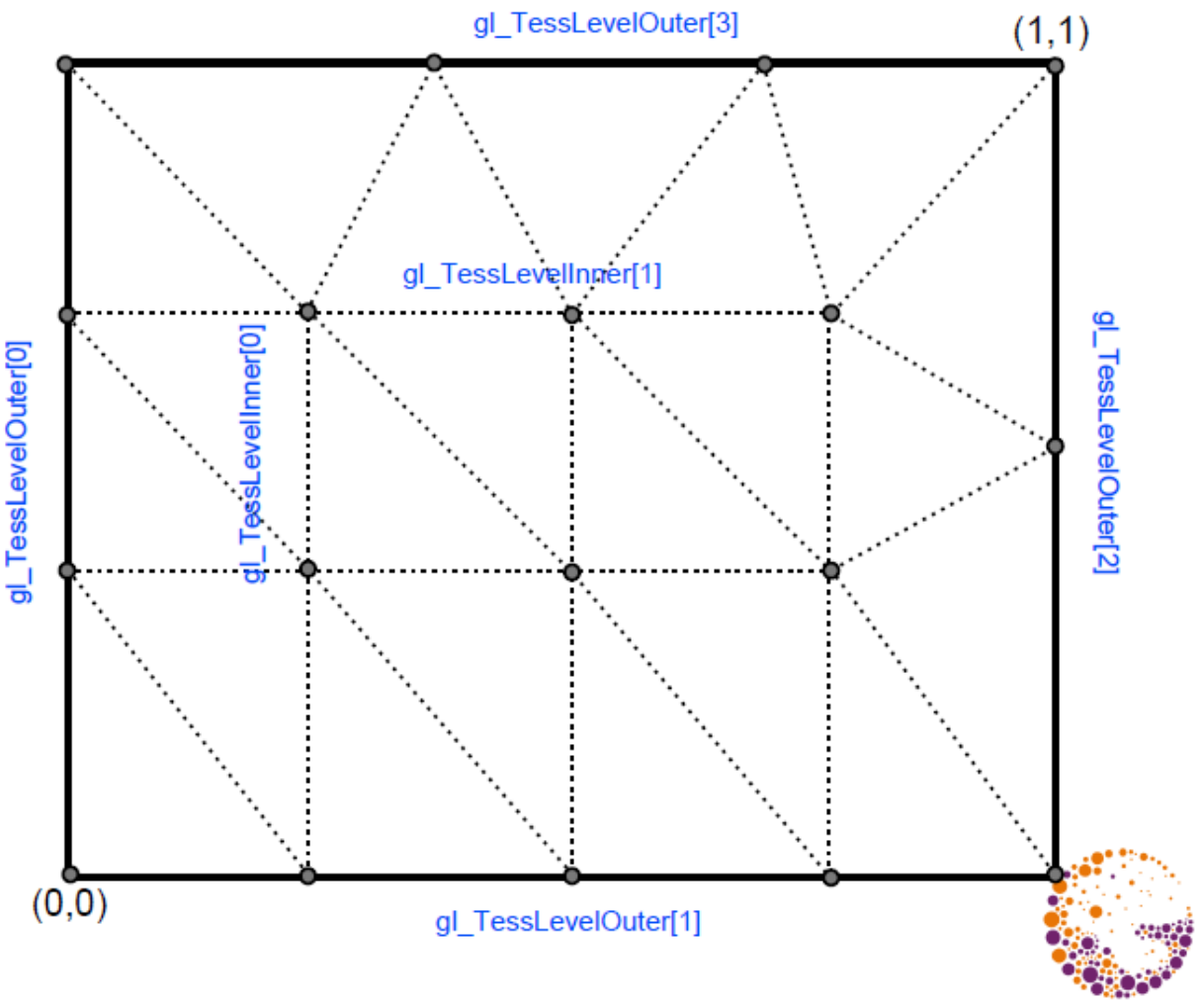
Generated primitives

Type	Param space	Tessellation factors
quad	unit square (u, v)	gl_TessLevelInner: 0 … 1 gl_TessLevelOuter: 0 … 3
triangle	barycentric (u, v, w)	gl_TessLevelInner: 0 gl_TessLevelOuter: 0 … 2
isolines	line (u, v) 'v' is constant for a line	gl_TessLevelOuter: 0 … 1

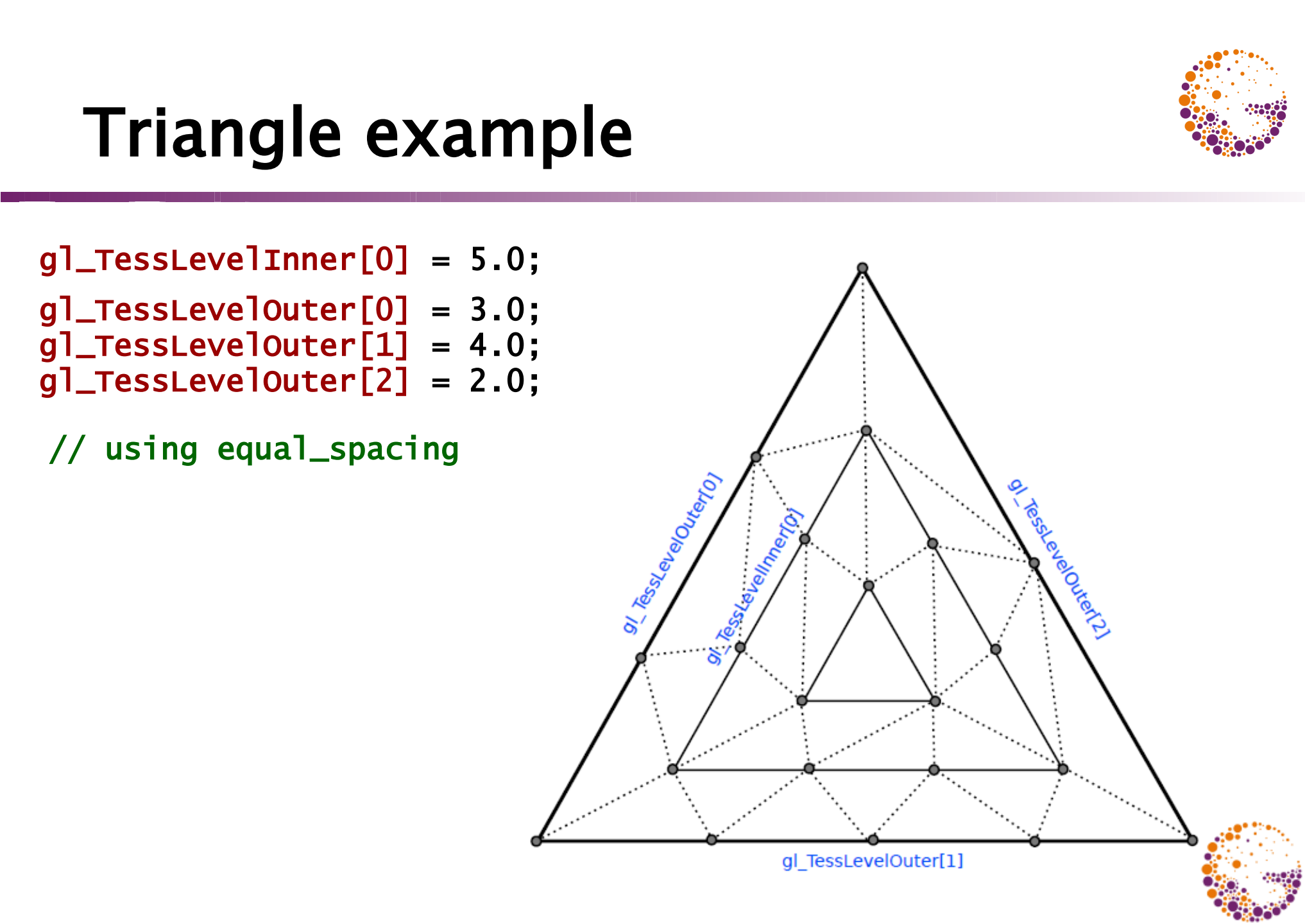
Quad example


```
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
```



Triangle example



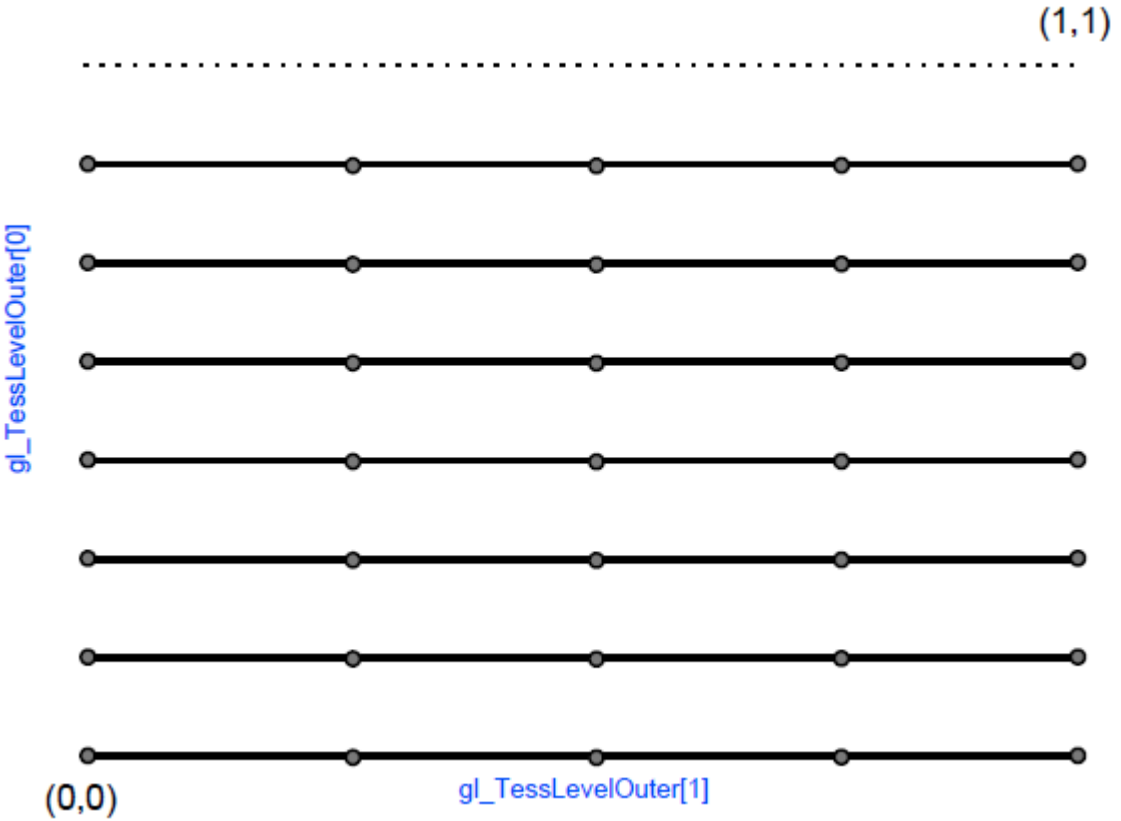
```
gl_TessLevelInner[0] = 5.0;
gl_TessLevelOuter[0] = 3.0;
gl_TessLevelOuter[1] = 4.0;
gl_TessLevelOuter[2] = 2.0;

// using equal_spacing
```

Lines example


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

// using equal_spacing
```



Tessellation control w/o shader

- in many cases we only need to copy data and configure Tessellation Primitive Generator..
- if input and output patch have the same vertex-number:

```
glPatchParameteri( GL_PATCH_VERTICES, vertices )
```

- inner and outer tessellation parameters (array lengths!):

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

Tessellation Evaluation Shader

- called independently for **every vertex** of the output mesh
- **input data** … patch
  - `gl_in[].gl_Position` …
  - `gl_TessCoord.xy`, `gl_TessCoord.xyz`
- **output** … vertex attributes (coordinate vector is mandatory)
  - `gl_Position`

Tessellation Evaluation Shader example

```
#version 400 core
layout ( quads, equal_spacing, ccw ) in;
uniform mat4 MV, P;  // model-view, projection
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;
}
```

Tessellation positions

- tessellation factors are **floating point** numbers!
  - **GL_MAX_TESS_GEN_LEVELS** (at least 64)

Tessellation mode	Allowed values	Subdivision
equal_spacing	[1, MAX]	**rounds up** to integer **equal** intervals
fractional_even_spacing	[2, MAX]	**rounds up** to **even int** **n-2** equal intervals **2** shorter
fractional_odd_spacing	[1, MAX-1]	**rounds up** to **odd int** **n-2** equal intervals, …

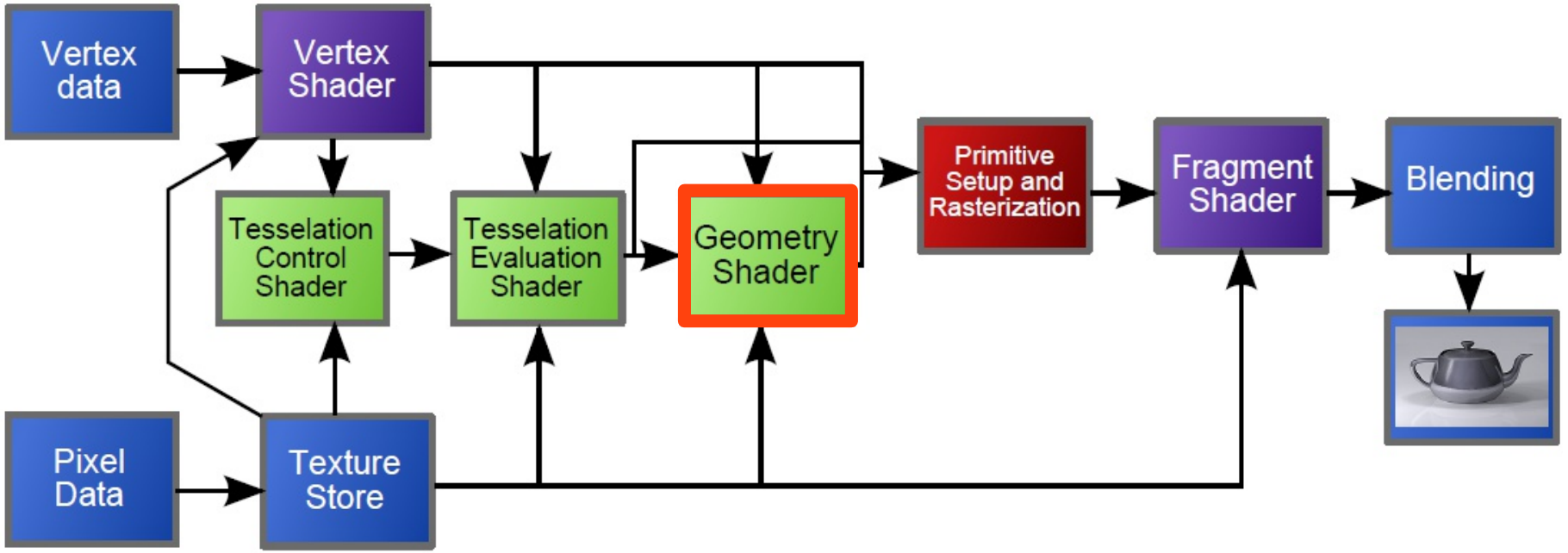
Primitive winding, Point mode

- default winding = counter-clockwise (**`ccw`**)
  - use '**`cw`**' for reverse direction
- there is an option for generating **points** instead of triangles
  - use '**`point_mode`**' in the '**`layout`**' directive

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

Geometry shader

- last optional stage before rasterizing stages
  - able to generate **new geometry – general approach**
    - no strict limits on output topology
  - **number** and **type** of output primitives



Geometry shader input

- **assembled primitives**
  (no strips or fans)
  - `points` (**1** v)
  - `lines` (**2** v)
  - `lines_adjacency` (**4** v)
  - `triangles` (**3** v)
  - `triangles_adjacency` (**6** v)
- access to the whole primitive
  - adjacency for continuity

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

in int gl_PrimitiveIdIn;

// only in OpenGL 4.0+
in int gl_InvocationID;
```

Geometry shader output

- possible **output primitives**
  - `points`
  - `line_strip`
  - `triangle_strip`
- output primitive type is **independent on input**
- **zero** or more output primitives
  - e.g. two triangle-strips, each of 100 triangles..
- implementation is **not** necessary **optimal** for massive geometry generation
  - that is what tessellation stages are for..

Geometry shader output

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

out int gl_PrimitiveID;
out int gl_Layer;

// only in OpenGL 4.0+
out int gl_ViewportIndex;
```

Geometry shader example

```
#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 shaders I

- **primitive generation**
  - *Geometry*: explicit control
  - *Tessellation*: only parameter control
- **topology**
  - *Geometry*: access to limited neighborhood only
  - *Tessellation*: patch design is up to programmer
- **source primitives**
  - *Geometry*: limited set
  - *Tessellation*: arbitrary (patch is general primitive)

Geometry vs. Tessellation shaders II

- **mesh cracks**
  - *Geometry*: problematic, should be handled carefully
  - *Tessellation*: automatic elimination (mesh is waterproof by design)

Sources

- Tomas Akenine-Möller, Eric Haines: ***Real-time rendering, 3rd edition***, A K Peters, 2008, ISBN: 9781568814247
- OpenGL Architecture Review Board: ***OpenGL Programming: The Official Guide to Learning OpenGL***, Addison-Wesley, latest edition (8th edition for the OpenGL 4.1)
- The Khronos Group: ***The OpenGL Graphics System: A Specification (Core/Compatibility profile)***, `http://www.opengl.org/registry/`