Realtime Computer Graphics on GPUs

Generate Geometry on GPU

Jan Kolomazník

Department of Software and Computer Science Education
Faculty of Mathematics and Physics
Charles University in Prague

Computer Graphics Charles University

Motivation

Why?

- CPU vs GPU – general vs. fast
- Tradeoff: computation vs. data transfer
- Geometry compression
- Adding details procedurally
- Adaptive smoothing (LOD)

Geometry Instancing

Why?

- Lots of models, same vertex data
  - Particle systems
  - Forrests
  - Armies
  - ...
- Different transformations, texture data, ...
- Many draw calls – bottleneck
  - Rendering – fast
  - Issuing draw commands – slow
- Instancing – draw multiple objects by single call

How?

- Replace standard draw calls by instanced versions (extra count parameter):
  - glDrawArraysInstanced()
  - glDrawElementsInstanced()
- In shaders build-in variable gl_InstanceID
  - From interval [0, count)
  - Used for indexing arrays of offsets, shifting texture coords, ...

OFFSETS FROM UNIFORMS – VS

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;

out vec3 fColor;

uniform vec2 offsets[100];

void main()
{
    vec2 offset = offsets[gl_InstanceID];
    gl_Position = vec4(aPos + offset, 0.0, 1.0);
    fColor = aColor;
}
```

- Slow setup
- Limited amount of data

OFFSETS FROM ATTRIBUTE – VS

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aOffset;

out vec3 fColor;

void main()
{
    gl_Position = vec4(aPos + aOffset, 0.0, 1.0);
    fColor = aColor;
}
```

INSTANCED VERTEX ATTRIBUTE

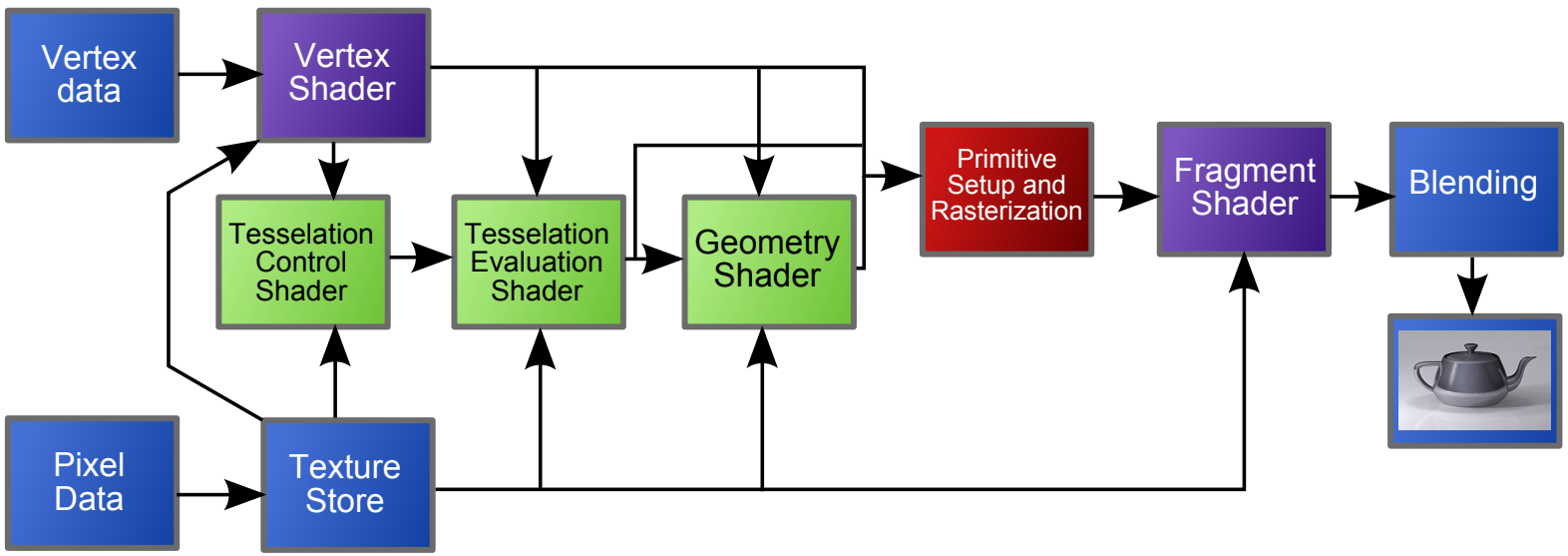
```
glEnableVertexAttribArray(2);
glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*)0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glVertexAttribDivisor(
   2 /*vertex attribute name*/,
   1 /*atribute divisor*/);
```

Attribute divisor:

- 0 – attribute updated for each vertex
- 1 – attribute updated for each instance
- n – attribute updated for each n instances

Tessellation Shaders

OpenGL Pipeline



Why?

- Adaptive subdivision based on a variety of criteria (size, curvature, etc.)
- Coarser models, but finer ones displayed (geometric compression)
- Detailed displacement maps without supplying equally detailed geometry
- Adapt visual quality to the required level of detail

Patch Primitive

- New *PATCH* primitive
- Fixed number of vertices:
  - `glPatchParameteri( GL\_PATCH\_VERTICES, num );`
  - Structure defined by shader implementor

Shader Organization

- Tessellation Control Shader (TCS)
  - Computes tessellation levels (fixed, distance to eye, screen space, hull curvature, ...
  - One invocation per output vertex
- Tessellation Primitive Generator (TPG)
  - Fixed-function
  - Generates predefined patterns in u-v-w barycentric coordinates
- Tessellation Evaluation Shader (TES)
  - Evaluates the surface in uvw coordinates
  - Interpolates attributes
  - Applies displacements

In OpenGL

```
glPatchParameteri( GL\_PATCH\_VERTICES, num );
GLuint tcs = glCreateShader( GL\_TESS\_CONTROL\_SHADER );
GLuint tes = glCreateShader( GL\_TESS\_EVALUATION\_SHADER ↩
    );
```

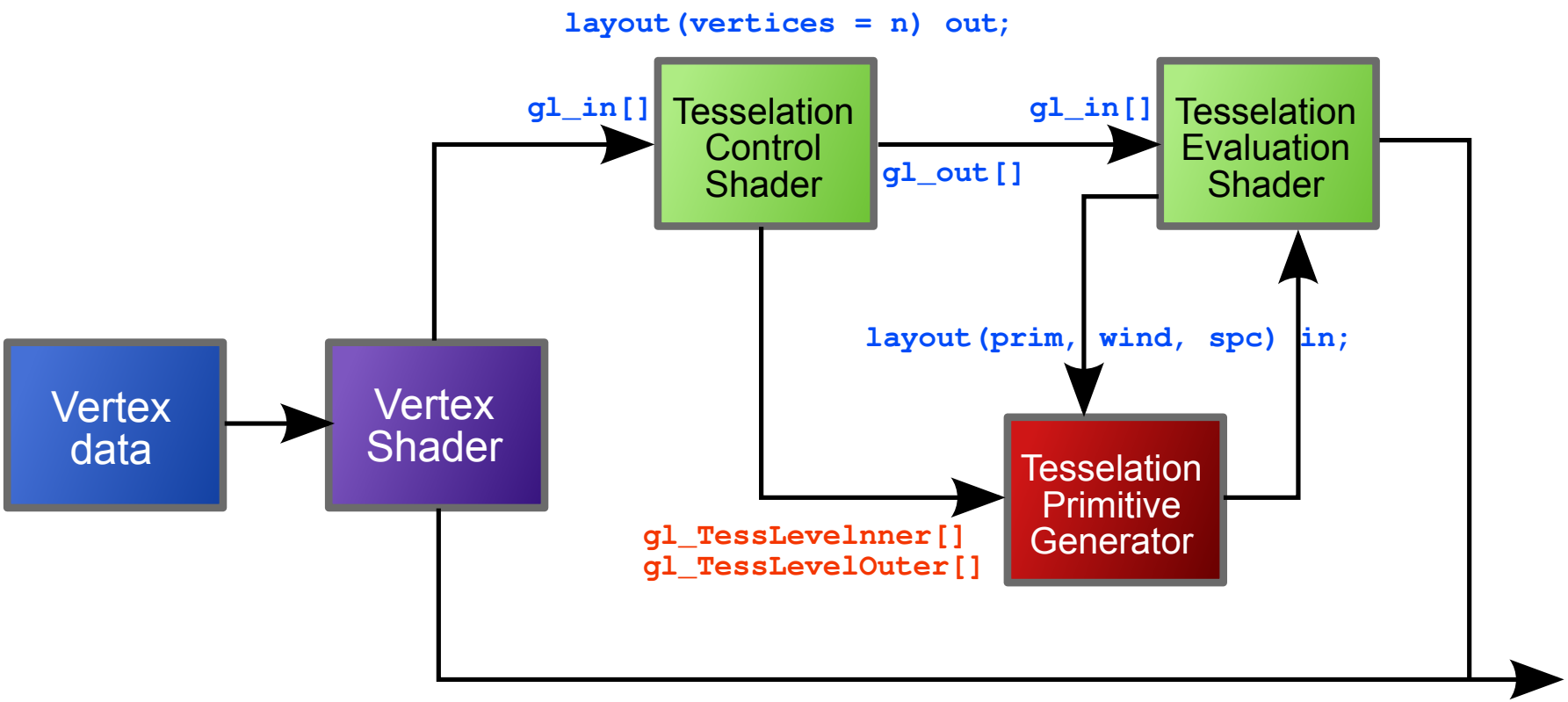
Data Flow During Tessellation

- Tessellation shader work on sets of vertices
  - Input/output data in arrays
- variable `gl_in[]` input data for both shaders
  - array length = `gl_in.length()`
- variable `gl_out[]` output parameters

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

- custom `in`/`out` attribute arrays
- `patch in`/`patch out` – per patch data

Data Flow During Tessellation



Tessellation Control Shader

- Output layout `vertices` – number of output vertices
- Control shader executed per input vertex
- Input and output – `patch`
- Access to all primitive attributes
- Can write only to its own output
  - `gl_InvocationID`
  - inter thread cooperation – synchronization (barrier())
- Main task – setup tesselation, recalculate the `patch`

EXAMPLE – TESSELLATION CONTROL SHADER

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 WITHOUT SHADER

- Lots of programs are just pass-through
- If input and output *patch* have same number of vertices – TCS can be autogenerated
  1. Specify number of vertices in input *patch*

     ```
     glPatchParameteri( GL_PATCH_VERTICES, NumVertices
     ```

  2. Specify inner and outer tesselations

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

PRIMITIVE GENERATOR

- Primitives generated by splitting parametric space
- 3 types of parametrization:

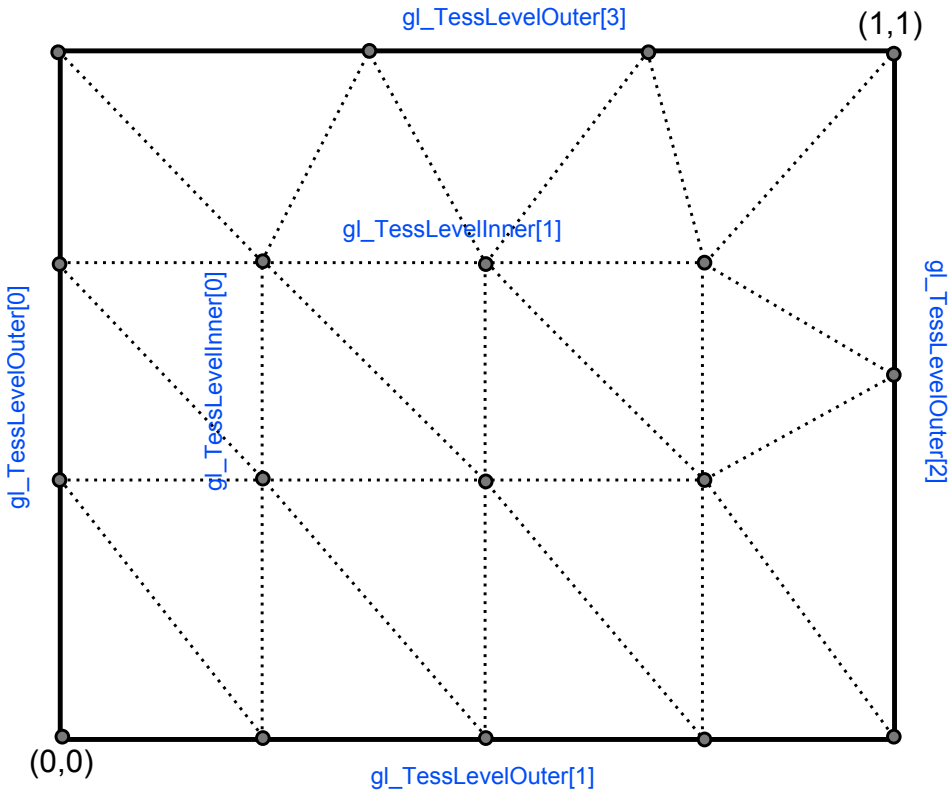
Type	Parametric space	Tessellation factors
quad	unit square: (u, v) u, v $\in$ [0, 1]	`gl_TessLevelInner`: 0 ... 1 `gl_TessLevelOuter`: 0 ... 3
triangle	barycentric: (u, v, w) u, v, w $\in$ [0, 1] u + v + w = 1	`gl_TessLevelInner`: 0 `gl_TessLevelOuter`: 0 ... 2
isolines	line: (u, v) u, v $\in$ [0, 1] v is constant on line	`gl_TessLevelOuter`: 0 ... 1

Quad Tessellation

parameters

```
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)
```

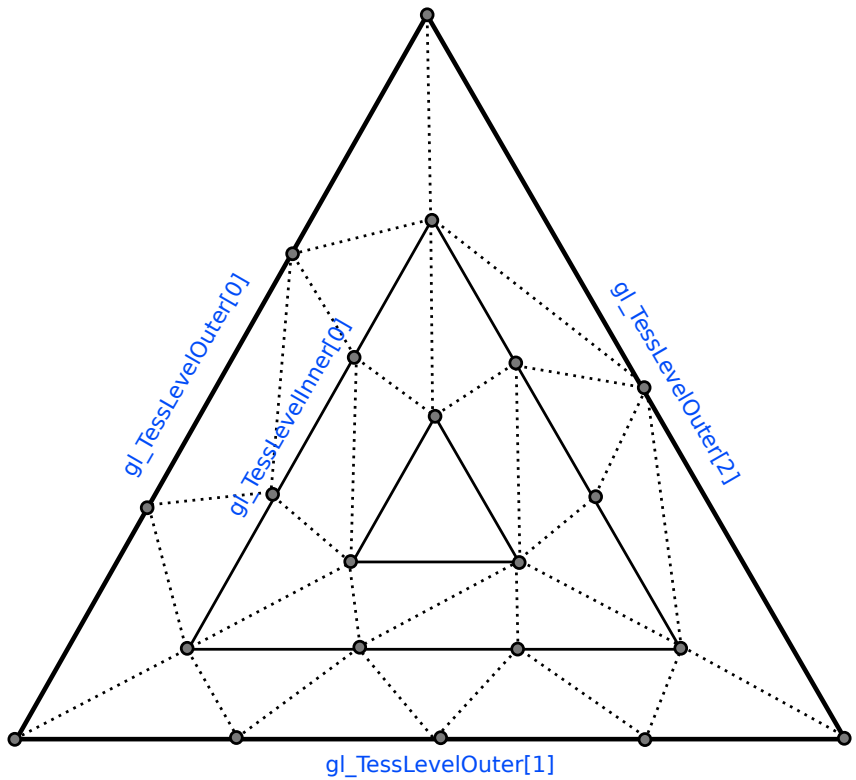


Triangle Tessellation

parameters

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

(using equal_spacing)
```

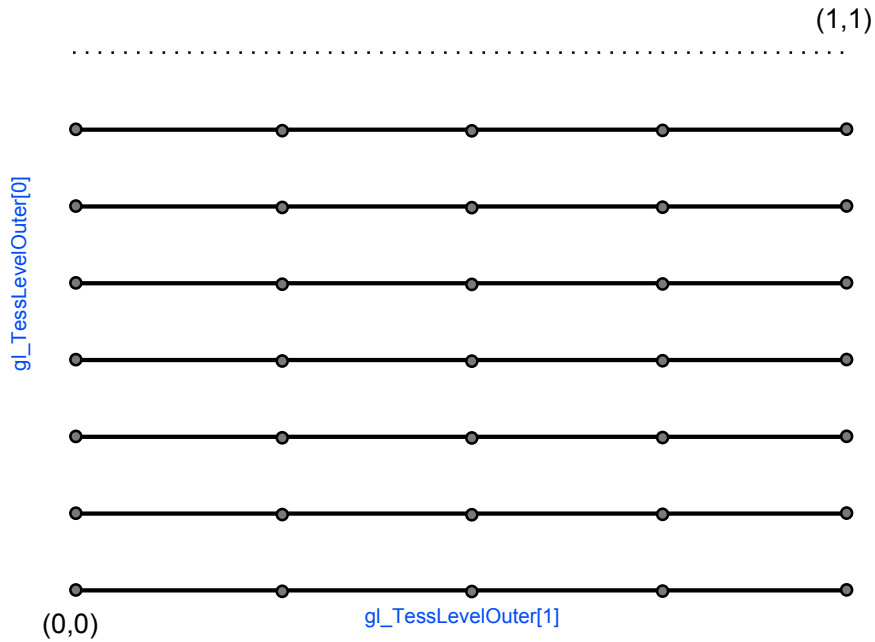


Isolines Tessellation

parameters

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

(using equal_spacing)
```



EXAMPLE – TESSELLATION EVALUATION SHADER

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;
}
```

Tessellation Position Control

- Tessellation factors – floating point numbers
- Each edge can be split into max `GL_MAX_TESS_GEN_LEVELS` – currently 64
- Different modes – different distribution of split points

Tessellation mode	**Range**
`equal_spacing`	[1, *max*]
`fractional_even_spacing`	[2, *max*]
`fractional_odd_spacing`	[1, *max-1*]

- `equal_spacing` integer splitting (rounds up), creates *n* same intervals
- `fractional_...` rounds to closest higher even/odd number, *(n-2)* intervals are same and 2 dependent on fractional part of the factor

Primitive Winding and Point Mode

- By default vertices organized by couter-clockwise direction (`ccw`)
  - Use `cw` for clockwise direction
- Instead of solid triangles can generate **points** – `point_mode` in `layout` directive

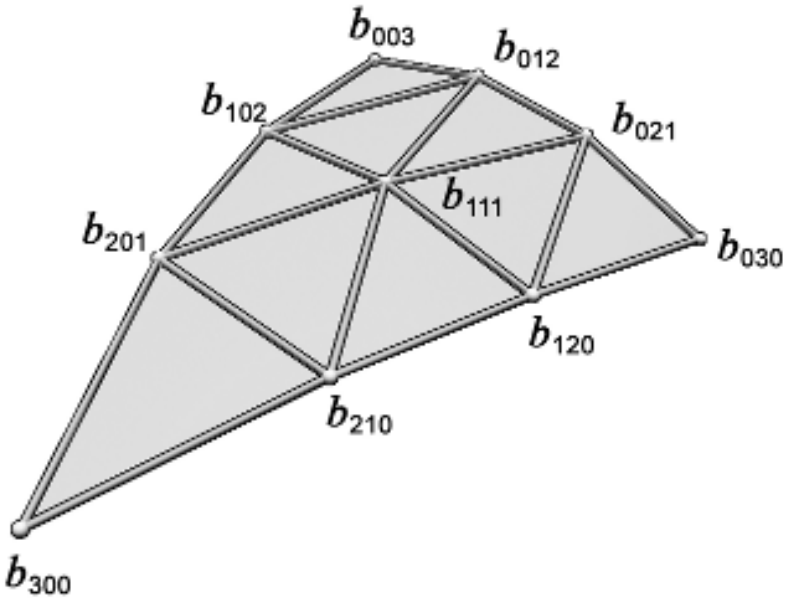
Sample layout in evaluation shader

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

Application – PN Triangles

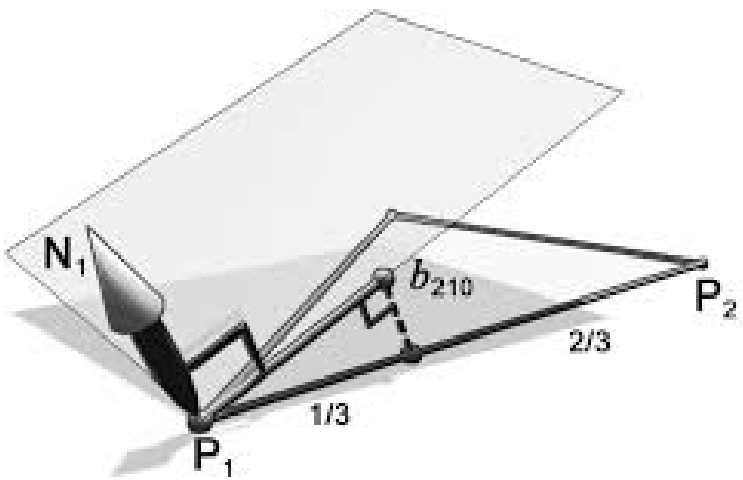
- Smooth triangular meshes
- Bezier triangle
- Surface control – interpolated normals (Point-normal triangles)

$$b(u,v,w) = \sum_{i+j+k=3} b_{ijk} \frac{3!}{i!j!k!} u^i v^j w^k$$



PN Triangles – Tangents

- Divide edges into 1/3
- Project to tangent plane defined by normal



TCS – PN Triangles

```
#version 410 core
// define the number of CPs in the output patch
layout (vertices = 1) out;
uniform int uOuter = 2, uInner = 2;
// attributes of the input CPs
in vec3 WorldPos_CS_in[];
in vec2 TexCoord_CS_in[];
in vec3 Normal_CS_in[];

struct OutputPatch
{
    vec3 WorldPos_B030;
    vec3 WorldPos_B021;
    vec3 WorldPos_B012;
    vec3 WorldPos_B003;
    vec3 WorldPos_B102;
    vec3 WorldPos_B201;
    vec3 WorldPos_B300;
    vec3 WorldPos_B210;
    vec3 WorldPos_B120;
    vec3 WorldPos_B111;
    vec3 Normal[3];
    vec2 TexCoord[3];
};

// attributes of the output CPs
out patch OutputPatch oPatch;
```

TCS – PN Triangles II

```
void CalcPositions()
{
    // The original vertices stay the same
    oPatch.WorldPos_B030 = WorldPos_CS_in[0];
    oPatch.WorldPos_B003 = WorldPos_CS_in[1];
    oPatch.WorldPos_B300 = WorldPos_CS_in[2];

    // Edges are names according to the opposing vertex
    vec3 EdgeB300 = oPatch.WorldPos_B003 - oPatch.WorldPos_B030;
    vec3 EdgeB030 = oPatch.WorldPos_B300 - oPatch.WorldPos_B003;
    vec3 EdgeB003 = oPatch.WorldPos_B030 - oPatch.WorldPos_B300;

    // Generate two midpoints on each edge
    oPatch.WorldPos_B021 = oPatch.WorldPos_B030 + EdgeB300 / 3.0;

    // ...

    // Project each midpoint on the plane defined by the nearest vertex and its normal
    oPatch.WorldPos_B021 = ProjectToPlane(oPatch.WorldPos_B021, oPatch.WorldPos_B030,
                                          oPatch.Normal[0]);
    // ...
    // Handle the center
    vec3 Center = (oPatch.WorldPos_B003 + oPatch.WorldPos_B030 + oPatch.WorldPos_B300) ↩
         / 3.0;
    oPatch.WorldPos_B111 = (oPatch.WorldPos_B021 + oPatch.WorldPos_B012 + oPatch.↩
         WorldPos_B102 +
                            oPatch.WorldPos_B201 + oPatch.WorldPos_B210 + oPatch.↩
                                 WorldPos_B120) / 6.0;
    oPatch.WorldPos_B111 += (oPatch.WorldPos_B111 - Center) / 2.0;
}
```

TCS – PN Triangles III

```
void main()
{
    // Set the control points of the output patch
    for (int i = 0 ; i < 3 ; i++) {
       oPatch.Normal[i] = Normal_CS_in[i];
       oPatch.TexCoord[i] = TexCoord_CS_in[i];
    }

    CalcPositions();

    // Calculate the tessellation levels
    gl_TessLevelOuter[0] = uOuter;
    gl_TessLevelOuter[1] = uOuter;
    gl_TessLevelOuter[2] = uOuter;
    gl_TessLevelInner[0] = uInner;
}
```

TES – PN Triangles I

```
void main()
{
    // Interpolate the attributes of the output vertex using the barycentric ↩
        coordinates
    TexCoord_FS_in = interpolate2D(oPatch.TexCoord[0], oPatch.TexCoord[1], oPatch.↩
        TexCoord[2]);
    normal = interpolate3D(oPatch.Normal[0], oPatch.Normal[1], oPatch.Normal[2]);

    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;
    float w = gl_TessCoord.z;

    float uPow3 = pow(u, 3);
    // ...

    WorldPos_FS_in = oPatch.WorldPos_B300 * wPow3 +
                     oPatch.WorldPos_B030 * uPow3 +
                     oPatch.WorldPos_B003 * vPow3 +
                     oPatch.WorldPos_B210 * 3.0 * wPow2 * u +
                     oPatch.WorldPos_B120 * 3.0 * w * uPow2 +
                     oPatch.WorldPos_B201 * 3.0 * wPow2 * v +
                     oPatch.WorldPos_B021 * 3.0 * uPow2 * v +
                     oPatch.WorldPos_B102 * 3.0 * w * vPow2 +
                     oPatch.WorldPos_B012 * 3.0 * u * vPow2 +
                     oPatch.WorldPos_B111 * 6.0 * w * u * v;

    gl_Position = u_projMat * u_viewMat * vec4(WorldPos_FS_in, 1.0);
}
```

Geometry Shaders

Overview

- last *optional* shader stage before rasterizer
- similar to tesselation
  - generating new geometry
  - similar results by differen principles
  - new primitives are generated directly in shader (no fixed primitive generator)

Geometry Primitives I

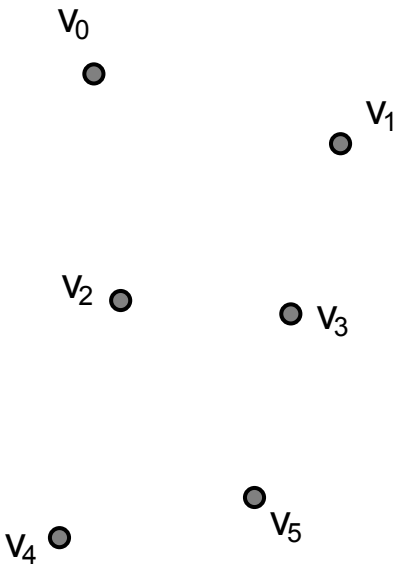


Figure: GL_POINTS

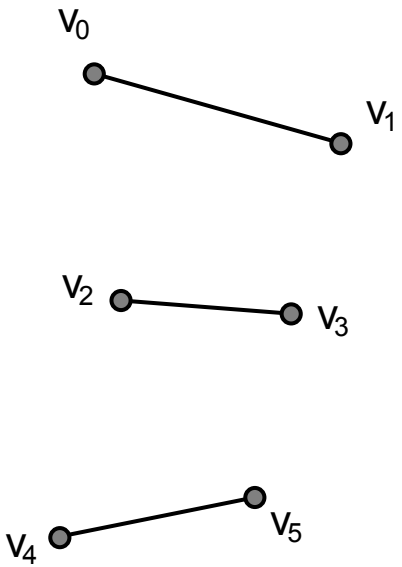


Figure: GL_LINES

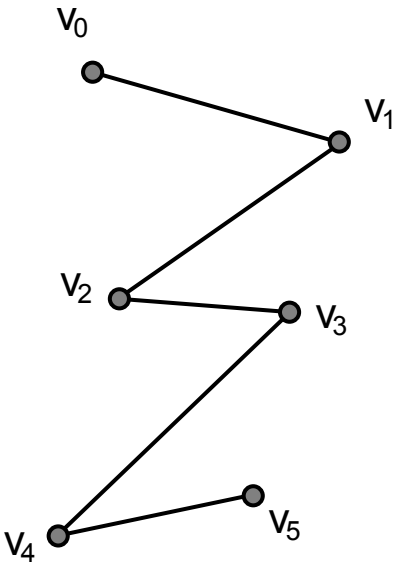


Figure: GL_LINE_STRIP

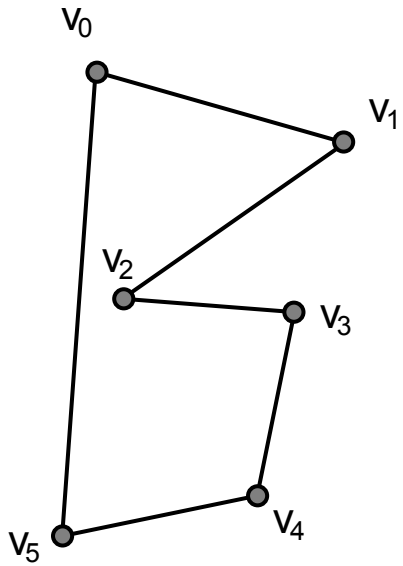


Figure: GL_LINE_LOOP

Geometry Primitives II

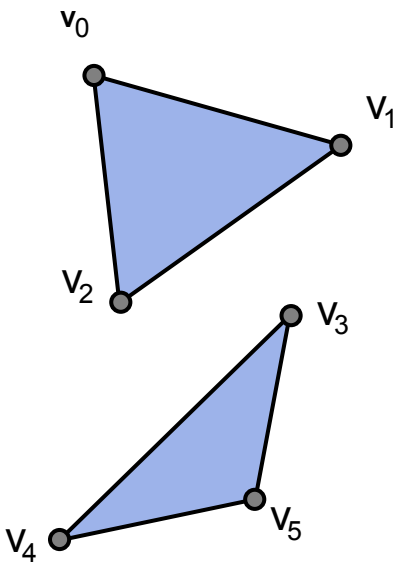


Figure:
GL_TRIANGLES

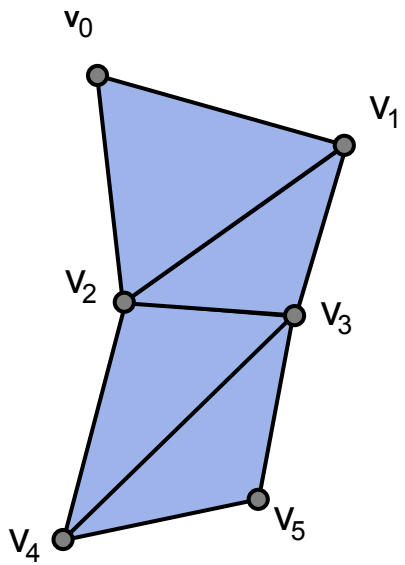


Figure:
GL_TRIANGLE_STRIP

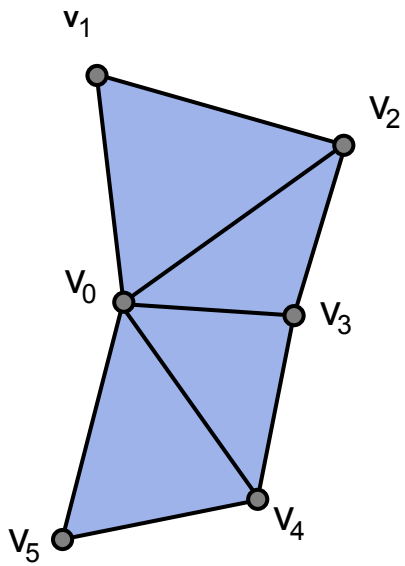


Figure:
GL_TRIANGLE_FAN

Geometry Primitives III

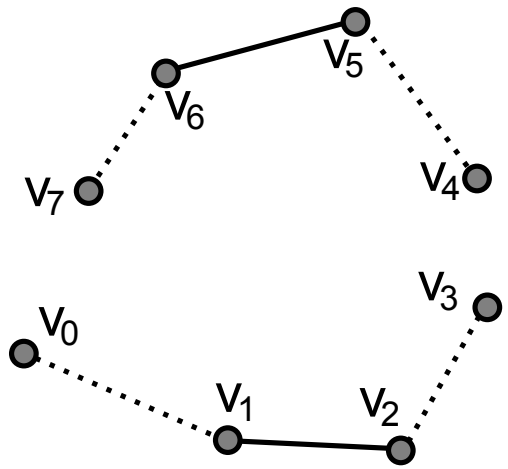


Figure: GL_LINES_ADJACENCY

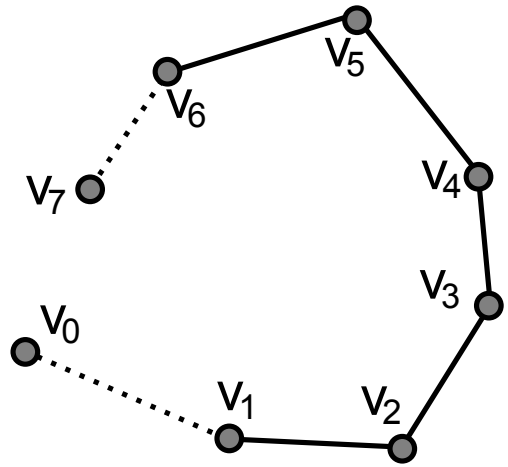


Figure: GL_LINE_STRIP_ADJACENCY

Geometry Primitives IV

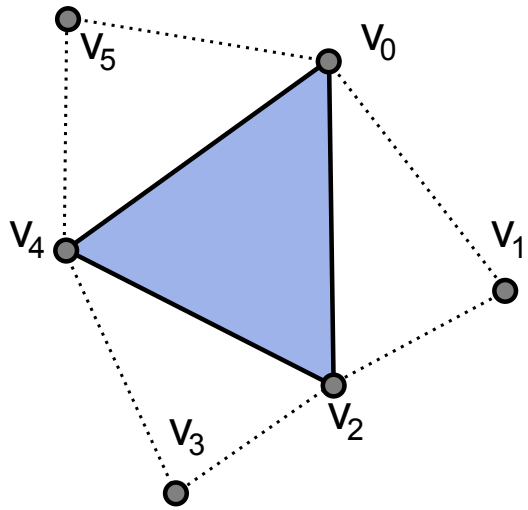


Figure:
GL_TRIANGLES_ADJACENCY

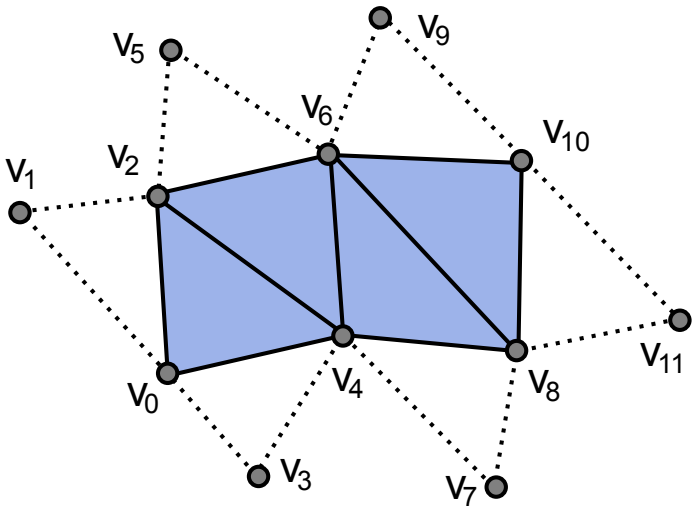


Figure:
GL_TRIANGLE_STRIP_ADJACENCY

Input

- *assembled* primitives (no *strips*, *loops*, or *fans*)
  - `points` ...1
  - `lines` ...2
  - `lines_adjacency` ...4
  - `triangles` ...3
  - `triangles_adjacency` ...6
- Access to information about the whole primitive
- New primitives with adjacency information

Builtin input variables

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

in int gl_PrimitiveIdIn;

// in OpenGL 4.0+,
// GS instancing
in int gl_InvocationID;
```

Output

- possible output primitives:
  - `points`
  - `line_strip`
  - `triangle_strip`
- types of input and output primitives are **independent**
  - input forgotten after shader execution
- output **0** or **more** primitives (up to implementation defined limit)

Output Production

1. Set vertex output attribs (gl_Position, ...)
2. Call EmitVertex()
3. Back to 1. to process next vertex, until primitive finished
4. Call EndPrimitive()
5. Back to 1. to process first vertex from next primitive, until all primitives generated

EXAMPLE

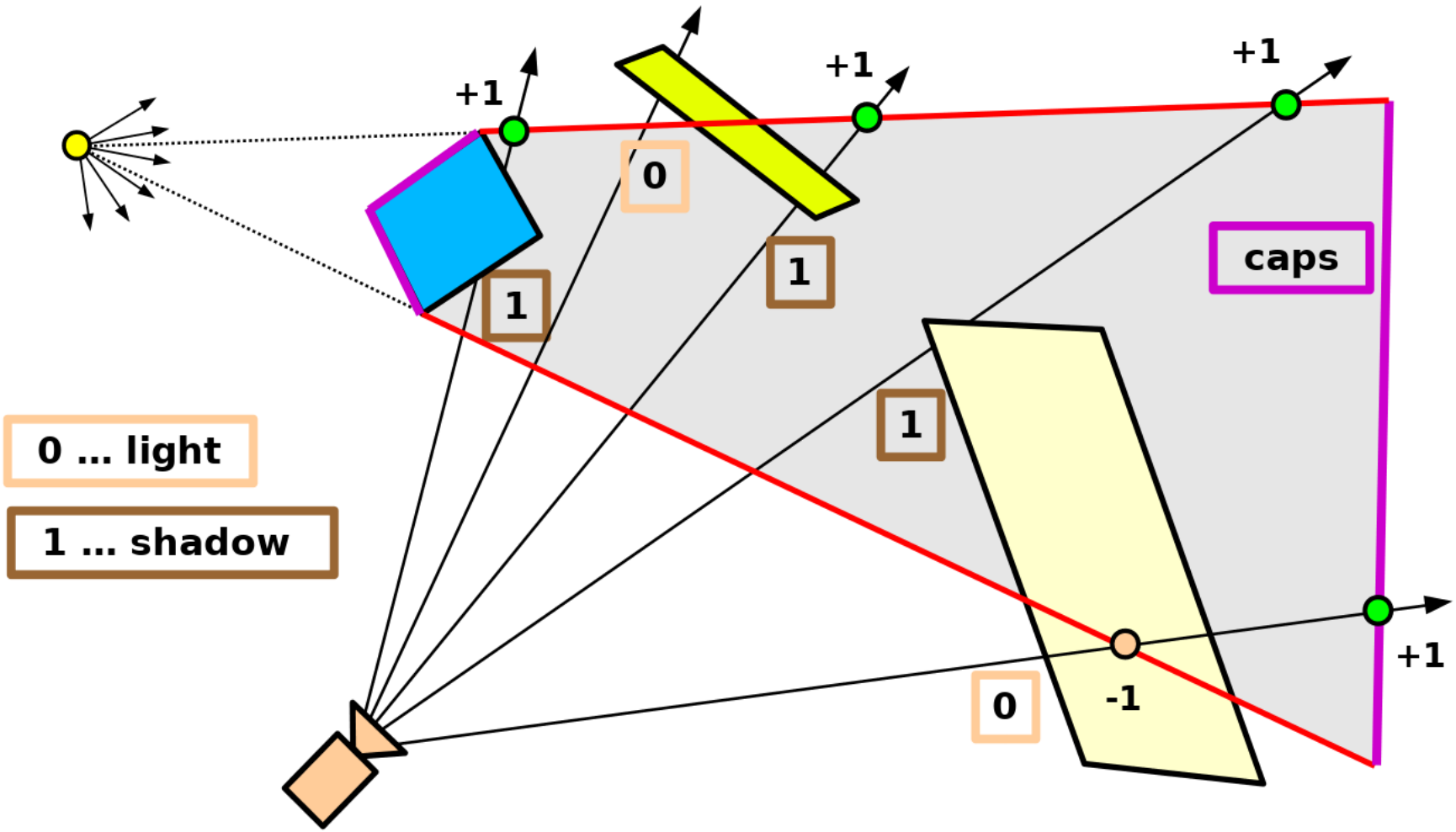
geometry shader

```
#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();
}
```

Application – Shadow Volume I

How to generate shadow volume?



Application – Shadow Volume II

Use triangles with adjacency:

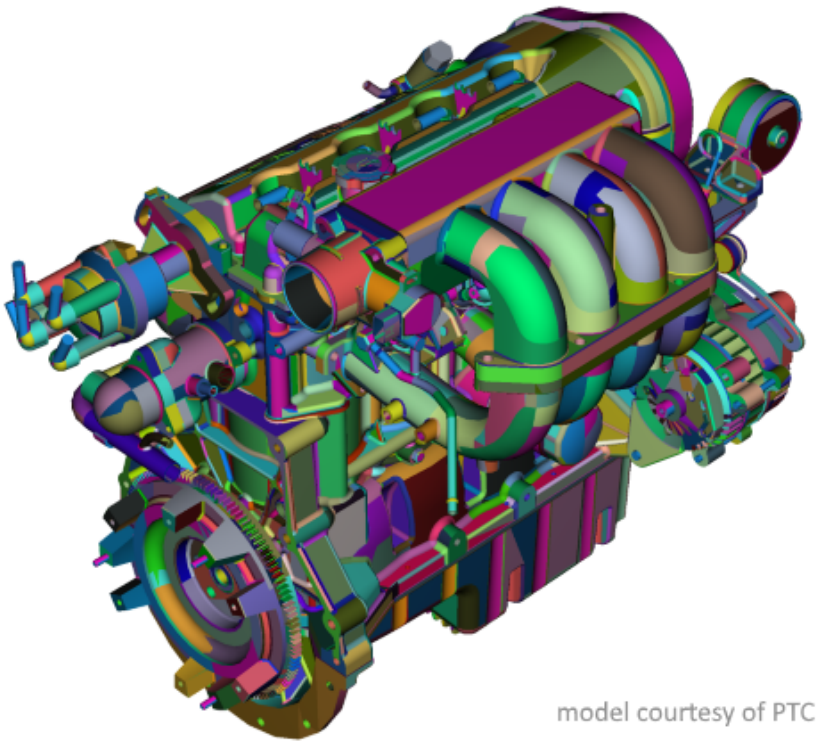
1. Render front cap
   - Pass through illuminated faces
2. Render back cap
   - Same polygons projected to infinity (depth clamping)
3. Render extruded silhouette
   - extrude edges separating illuminated and shadowed faces (compare $dot(N, L)$)

Mesh Shaders

Why?

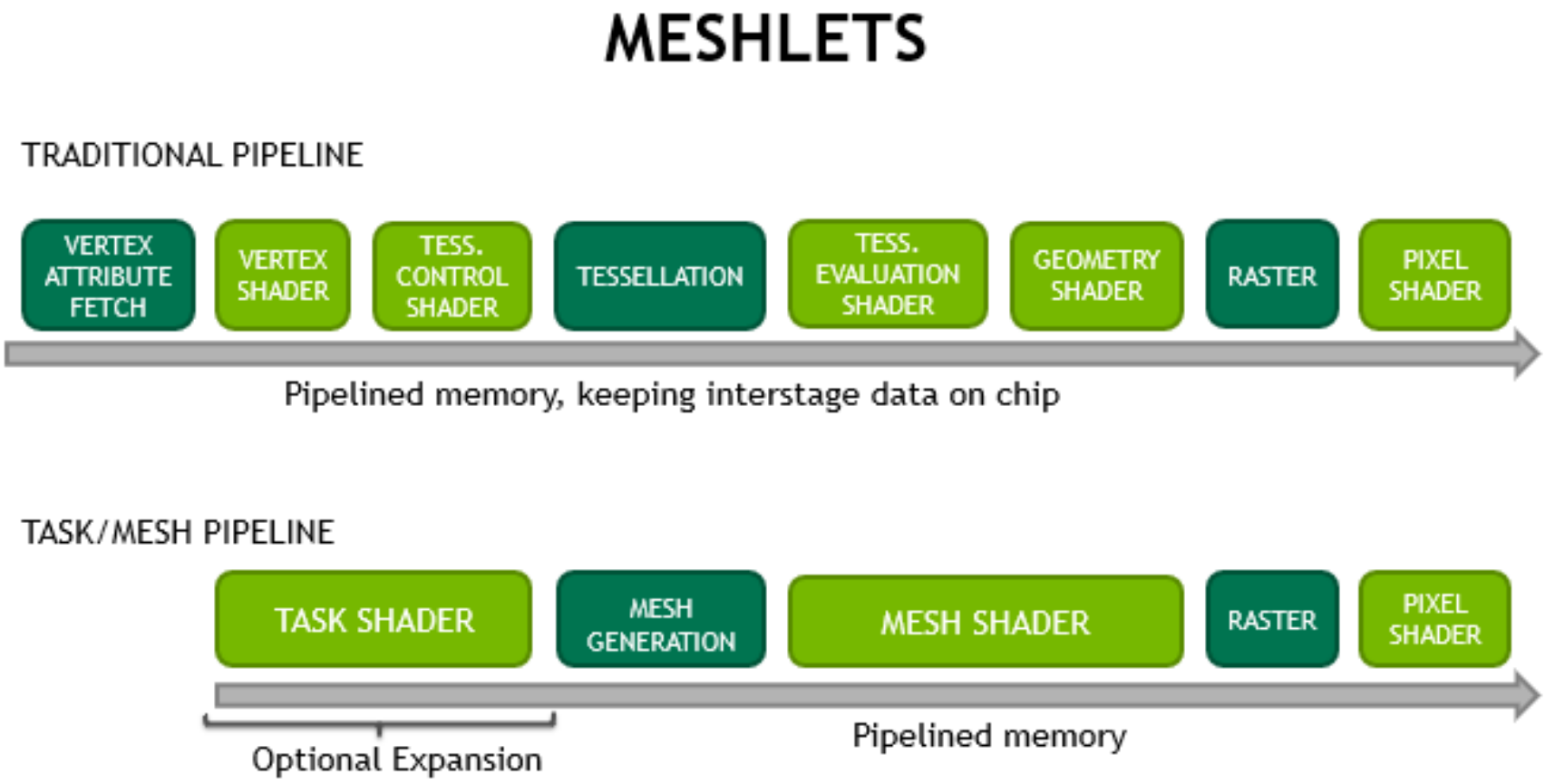
- Tessellation and geometry shaders are limited:
  - Tessellation in function
  - Geometry in speed
- Mesh shaders – convergence to compute shaders
  - Better hardware saturation
  - More flexible

MESHLETS



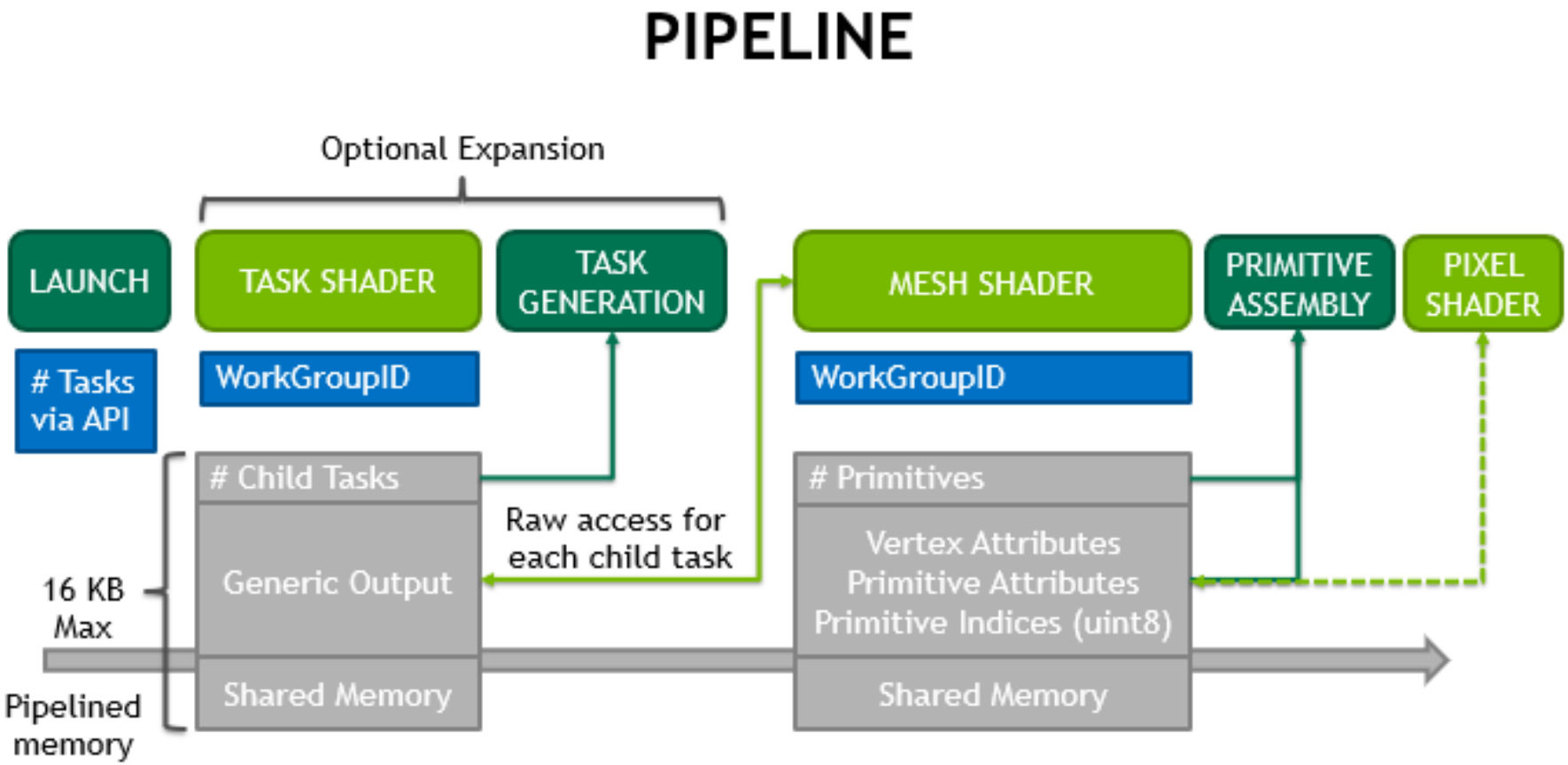
model courtesy of PTC

PIPELINE


MESHLETS
TRADITIONAL PIPELINE
VERTEX ATTRIBUTE FETCH
VERTEX SHADER
TESS. CONTROL SHADER
TESSELLATION
TESS. EVALUATION SHADER
GEOMETRY SHADER
RASTER
PIXEL SHADER
Pipelined memory, keeping interstage data on chip
TASK/MESH PIPELINE
TASK SHADER
MESH GENERATION
MESH SHADER
RASTER
PIXEL SHADER
Optional Expansion
Pipelined memory

STRUCTURE

- Task shader
- Mesh shader



COMPARISON

Shader		Thread Mapping	Topology
Vertex Shader	No access to connectivity	1 Vertex	No influence
Geometry Shader	Variable output doesn't fit HW well	1 Primitive / 1 Output Strip	Triangle Strips
Tessellation Shader	Fixed-function topology	1 Patch / 1 Evaluated Vertex	Fast Patterns
Mesh Shader	Compute shader features	Flexible	Flexible within work group allocation