Computer Graphics 1

Ludwig-Maximilians-Universität München
Summer semester 2020

Prof. Dr.-Ing. Andreas Butz
lecture additions by Dr. Michael Krone, Univ. Stuttgart

https://commons.wikimedia.org/wiki/File:Stanford_bunny_qem.png

Chapter 4 – OpenGL

- Motivation
- OpenGL Core
- OpenGL Shaders & Shading Language
- OpenGL Objects
- Geometry
- Textures
- Framebuffer
- Geometry Shaders

Motivation

- CG1 class presents the concepts for (more or less) realistic 3DCG
- Optimization for running time is partially ignored
- …but 3DCG is most interesting in real time!
  - Fast image generation (rendering)
    - Animation in real time
  - Interaction with 3DCG
    - Exploration
    - Selection
    - Parameter adjustment

www.idsoftware.com

This chapter was adapted by Dr. Michael Krone from the slides of Prof. Thomas Ertl und Dr. Guido Reina (VISUS, Univ. Stuttgart)

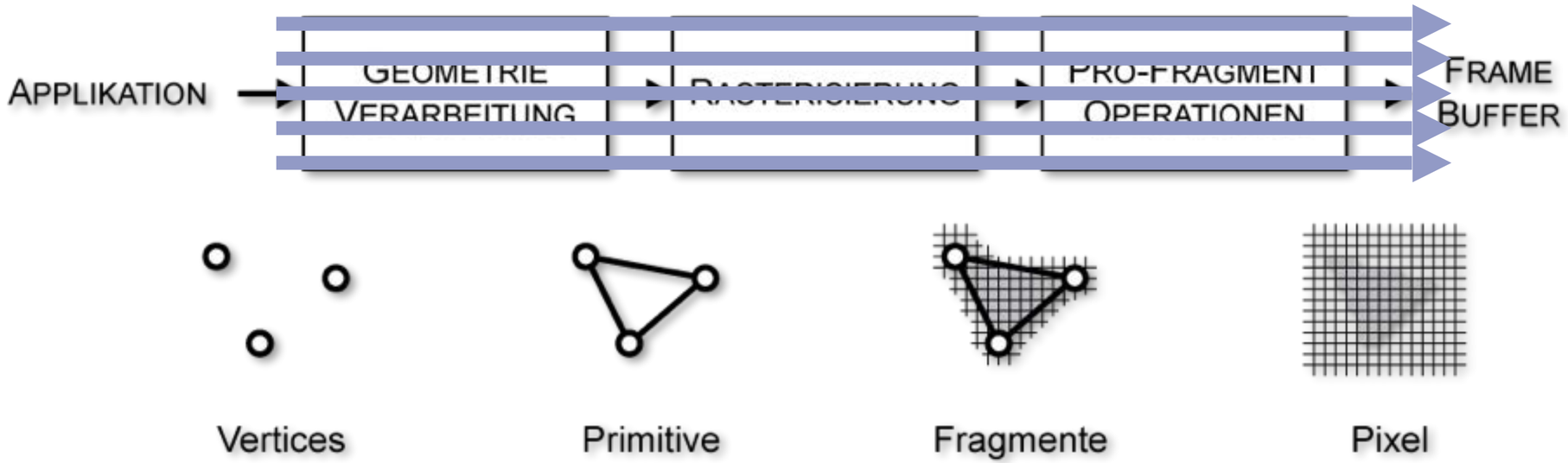
Motivation – Modern Realtime Graphics

Copyright: Quixel 2019

Motivation – Modern Realtime Graphics

Interactive 3DCG

- Rasterization (using a depth buffer) is efficient: Vertices and triangles all pass (independently) the same processing steps
- Concept of the graphics pipeline



- **Consequences or disadvantages:**
  - normally only local illumination + shadows
  - special effects + global illumination only via multiple passes through pipeline
  - Access to graphics hardware via APIs such as wie OpenGL, Direct3D, …

OpenGL – History and development

- 1983 - 1992: SGI Graphics Library (GL)
  - basically proprietary, 3D Rendering + input, windows, menus, text, …
- OpenGL (www.opengl.org)

  - Defined by Architecture Review Board (ARB), Khronos Group (2000)
    - SGI, IBM, Qualcomm, Imagination, Intel, Microsoft, HP, NVIDIA, ATI, Apple, 3Dlabs, …, Pixar, Lucasfilm, Valve, …
  - 1992: OpenGL 1.0 (fixed-function pipeline)

  - …
  - 2017: OpenGL 4.6 (modern, programmable pipeline)
- OpenGL ES: OpenGL for mobile and embedded systems
  - OpenGL ES 2.0 based on OpenGL 2.0 (2004)
  - OpenGL ES Shading Language 1.00

  - WebGL 1.0 is based on OpenGL ES 2.0
  - WebGL 2.0 is based on OpenGL ES 3.0 (2012)

    - three.js utilizes WebGL

Chapter 4 – OpenGL

- Motivation
- OpenGL Core
- OpenGL Shaders & Shading Language
- OpenGL Objects
- Geometry
- Textures
- Framebuffer
- Tesselation & Geometry Shader

What is OpenGL?

3D Rendering API: OpenGL

- Platform-, hardware- and language-independent programming interface for 3D graphics hardware
  - ~200 functions for defining geometry and executing typical 3DCG tasks
  - Independent of windowing system (no events, windows, menus, …)
- OpenGL implements a graphics pipeline
  - geometric primitives: points, lines, triangles, (up to OpenGL 3 also Quads)
  - buffers and textures
  - programmable processing steps in the pipeline (Shaders)
    - freely programmable processing of geometry, shading, tesselation, …

OpenGL – more properties

- State machine
  - controls and configures processing in the pipeline
  - example states: textures, shader, depth test, blending, …

- Low level API:
  - no higher concepts for modeling or animation,
  - no scene graph

- Extensions allow access to particular capabilities of a GPU

- GL Shading Language (GLSL) for programmable GPUs

OpenGL – a very first taste ;-)

- OpenGL is a (hierarchical) state machine
  - a state remains valid until it is explicitly changed
  - objects might have their own state/parameters (see *Direct State Access*)

- Setting OpenGL states

```
glEnable( ... );
glDisable( ... );
gl*( ... );          // various specific states
```

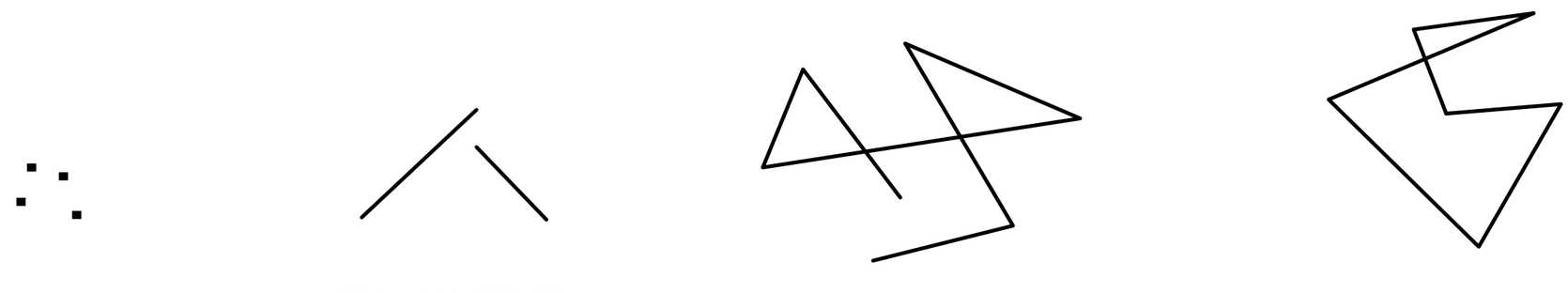
- Reading OpenGL states

```
glIsEnabled( ... );
glGet*( ... );     // specific states
```

  - to read the current state, the pipeline might have to run empty 

OpenGL: Geometric Primitives

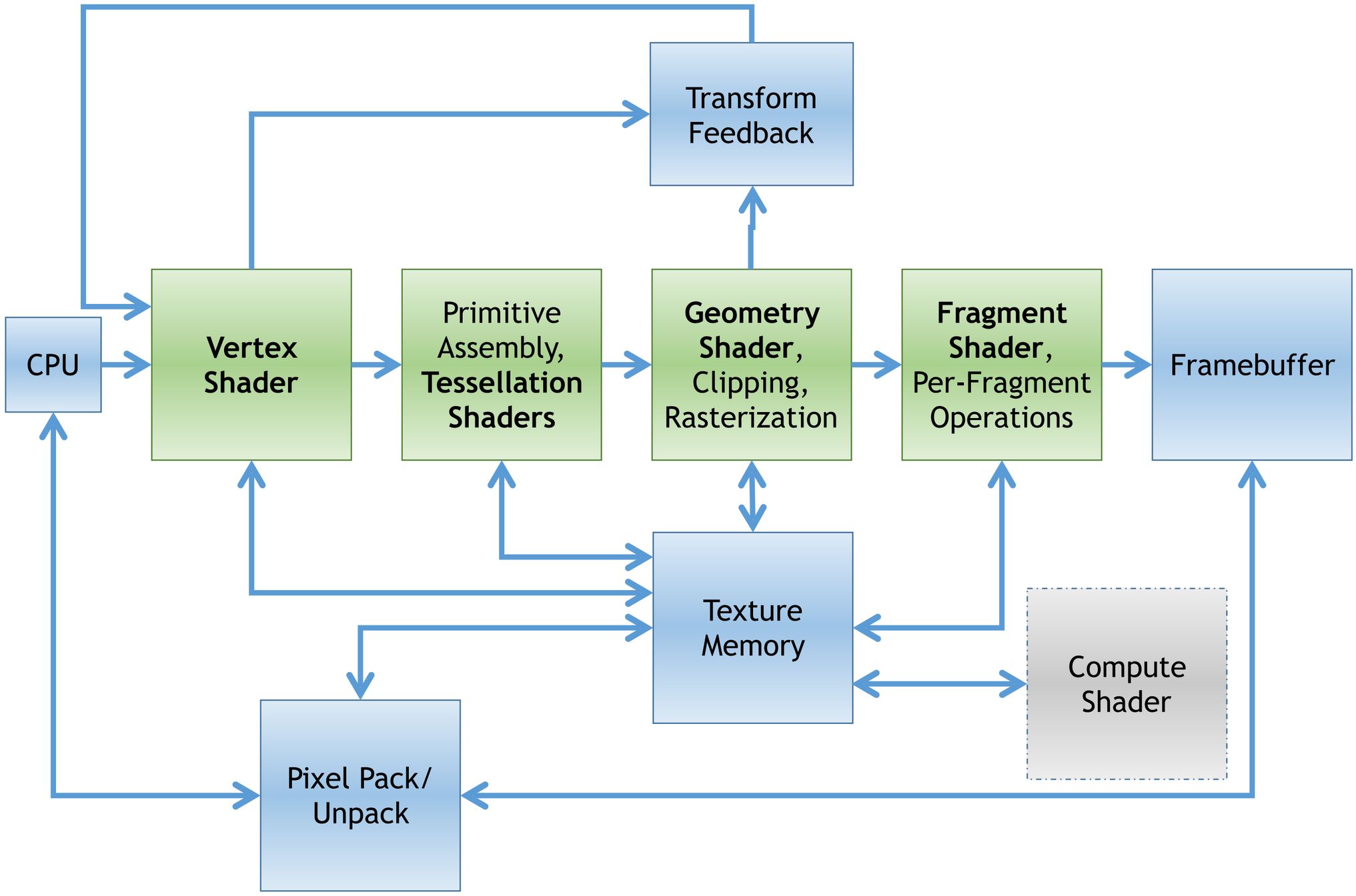
- all geometric primitives are defined by vertices in homogeneous coordinates (x, y, z, w).
  - if the homogeneous coordinate w is not specified, it is w=1



GL_POINTS **GL_LINES** **GL_LINE_STRIP** **GL_LINE_LOOP**

GL_TRIANGLES **GL_TRIANGLE_STRIP** **GL_TRIANGLE_FAN** **GL_PATCHES**

OpenGL(4.5)-Pipeline

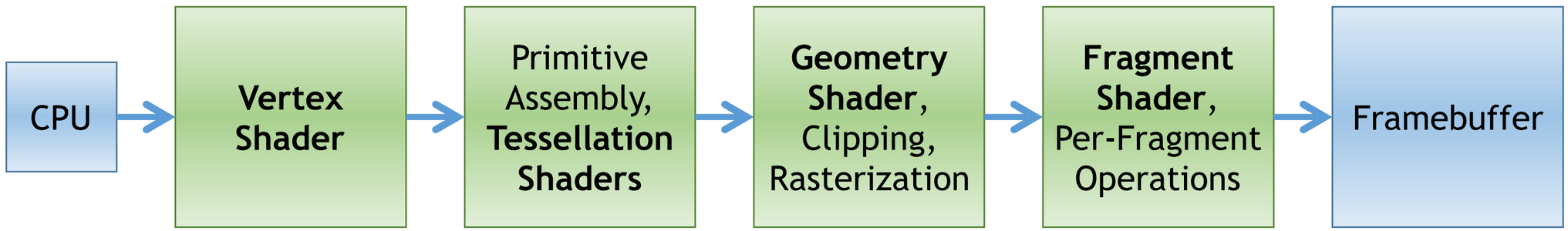


Chapter 4 – OpenGL

- Motivation
- OpenGL Core
- OpenGL Shaders & Shading Language
- OpenGL Objects
- Geometry
- Textures
- Framebuffer
- Tesselation & Geometry Shader

What is a Shader?

- Code for one of the programmable steps of the graphics pipeline
- 5(+1) types: vertex-, tessellation-control-, tessellation-, evaluation-, geometry- and fragment-shader (+ compute-shader)
- Basic structure and capabilities similar for all types
  - operation and functions are identical
  - semantics and layout of in- and output data varies
  - problems can be solved at various steps, but mostly only one specific shader makes sense



- Mainly high level languages, e.g., GLSL (OpenGL), HLSL (Direct3D)
- In current OpenGL (3+) and WebGL nothing works without vertex- and fragment-shaders (no output!)

OpenGL Shading Language (GLSL)

- SIMD – Single Instruction, Multiple Data
  - multiple data units (e.g., Vertex-Pos.) treated the same way simultaneously
  - imagine 1 processor for each Vertex/Primitive/Fragment
- Language similar to C
  - with particular data types and instructions
  - …and some restrictions
- Basic data types:
  - **float**, **double**, **int**, **bool**
- Vectors with 2, 3 or 4 components
  - **vec**{2.0, 3.0, 4.0}, **dvec**{2.0, 3.0, 4.0}, **bvec**{true, false, true}, **ivec**{2, 3, 4}
- Matrices from 2x2 to 4x4 also asymmetrical
  - **[d]mat2, [d]mat3, [d]mat4, [d]mat2x3, ...**
- Arrays (just one-dimensional, can be declared flexibly but fixed length)
  - **float a[5] = float[](3.4, 4.2, 5.0, 5.2, 1.1);**

GLSL: Vectors and Swizzling

- Components of a vector are accessed like structs in C

```
vector.[xyzwrgbastpq]+
```

- Coordinates (xyzw), color channels (rgba) and texture coordinates (stpq) are synonymous and can be mixed arbitrarily

```
vec3 col; col.rgb == col.xgp == col.stp;   // all equivalent
```

- Change of position and replication is possible

```
vec4 g; g.wxyz = in_pos.yyxx;
```

- Number and type of components have to match

```
vec2 g; vec4 f; f = g.xxxz;                          // wrong!
```

GLSL Qualifiers

- …define the purpose and usage of a variable
- **`const:`** constant, defined at compile time
- **`in:`** input attribute of current shader (optional, default)
- **`out:`** output attribute of current shader
- **`uniform:`** read-only
  - global variables: constant per Draw Call
    - set by applications!
    - can also come from buffers
- **`Buffer:`** global memory, readable and writable
- **`shared:`** only in compute shader, also writable

GLSL Control Flow

- **if/else, for, while, do/while**
  - **continue**: jumps to the next iteration
  - **break**: ends the loop
  - Don’t assume that the control flow in the program has the same runtime behavior as on the CPU/GPU! (however, semantics is identical!)

    - Example: in an **if/else** statement both paths might be executed in parallel, and then in the end only the ‚right‘ one (according to the conditional clause) will have an effect, caused by masking of the variables in the ‚wrong‘ path.
- **discard**
  - only in fragment shaders
  - ends the shader without writing to the depth/frame buffer
    - also here: the code might be fully executed but then the result ignored

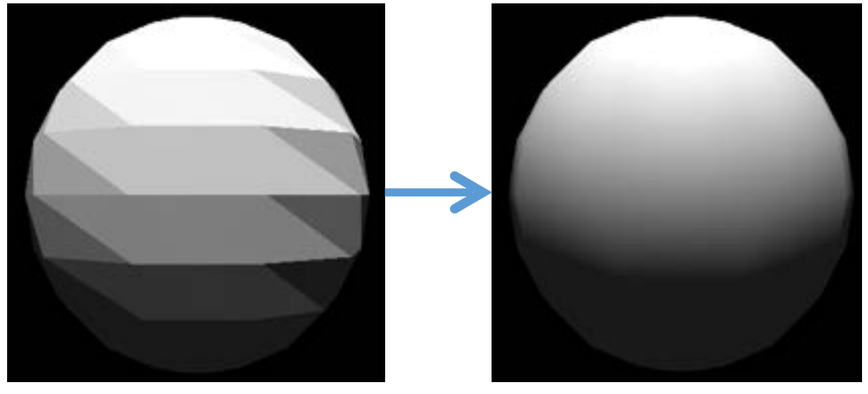
GLSL-Functions

- Shader is structured by functions
  - mandatory: **void main()**
- Own functions:
  - **in**: input parameter (optional)
  - **out**: return value (or return statement)
  - **inout**: both in- and output

```
void computeLighting(vec3 lightPos, vec3 normal, out float diffuse, out float specular) {
…
}
```

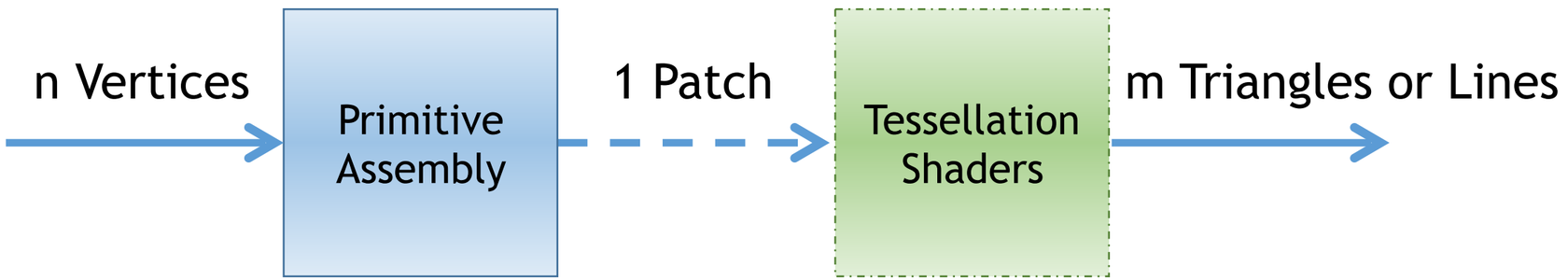
- Functions can be **overloaded**
- **Recursion** is (still!) **not allowed**

Geometry Processing: Vertex Shader



- Transformation of single vertices and their attributes (e.g., normals, …)
  - no creation of new vertices
  - no deletion of vertices (except by clipping, see later…)
- Computing of all attributes which are valid per vertex
  - saves computation in comparison to fragment shader
  - e.g., illumination per vertex (old fashioned)
- Setting attributes that will be interpolated per fragment
  - e.g., normal for illumination per vertex

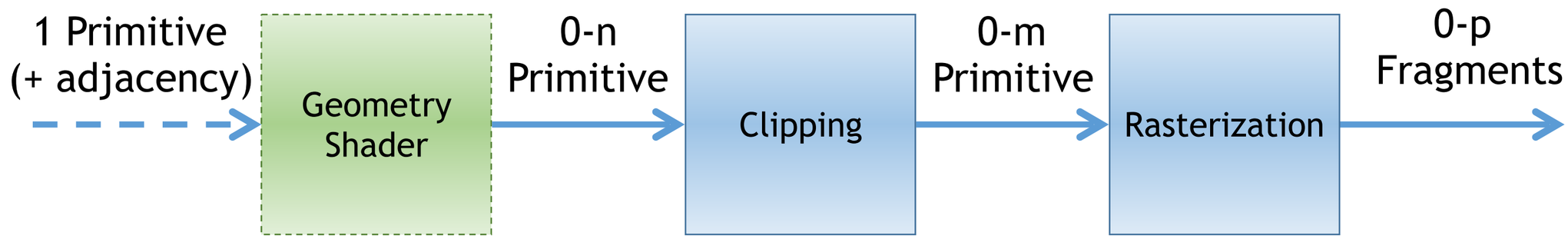
Geometry Processing: Assembly + Tessellation



- Primitive Assembly composes primitives from vertices (non programmable):
  - **`GL_POINT, GL_LINE, GL_TRIANGLE, GL_LINE_STRIP`** etc.
  - **`GL_PATCH`** as an input primitive for tessellation shaders

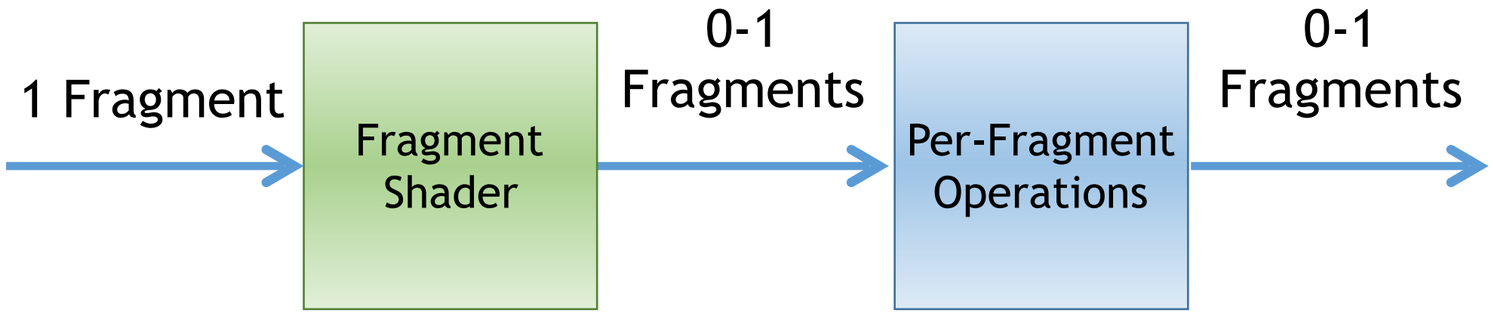
- Tessellation (parallel per primitive):
  - Tessellation Control Shader determines the subdivision
  - Tessellator executes the subdivision (non programmable)
  - Tessellation Evaluation Shader cats on the resulting geometry

Geometry Processing: Geometry Shader etc.



- Geometry Shader
  - works on arbitrary primitives, sequentially per primitive
  - Can extend, delete or modify primitives (points to triangles etc.)
  - Per primitive, only limited Attributes can be output
- Clipping (to viewing volume)
  - discards invisible point primitives
  - modifies lines and triangles by intersection with the viewing volume
- Perspective division
- Rasterization generates fragments for the output buffer

Fragment Processing



- Fragment Shader
  - allows computation per pixel for the output buffer
    - setting output color
    - per-pixel lighting/shading
    - computation on data within the primitive, e.g.,
      - volume rendering, implicit surfaces, glyphs
  - input attributes are interpolated within the primitive (unless switched off)
  - fragments can be discarded: **`discard`**
- fragment operations: tests, blending etc. (more later)

Chapter 4 – OpenGL

- Motivation
- OpenGL Core
- OpenGL Shaders & Shading Language
- OpenGL Objects
- Geometry
- Textures
- Framebuffer
- Tesselation & Geometry Shader

OpenGL Objects

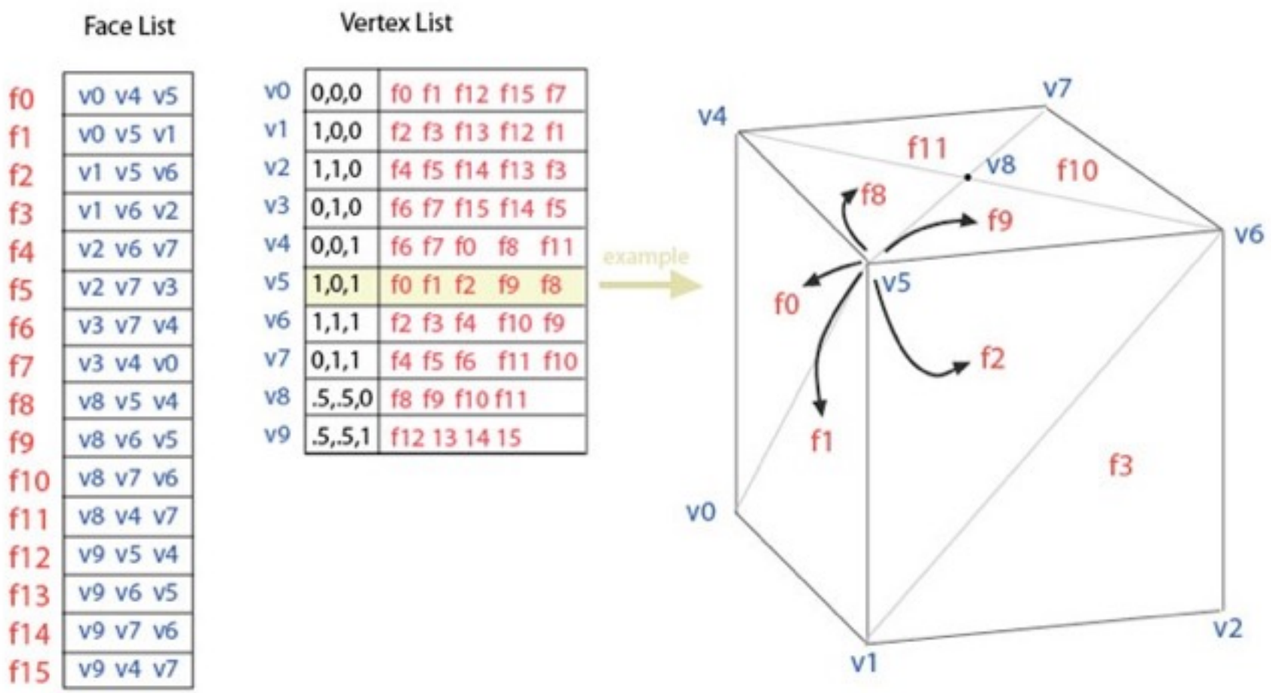
- Textures, geometry, shaders etc. are OpenGL objects
  - Not objects in the OOP sense!!
    - No memory pointers, but handles
- Creating, initializing and deleting objects
  - Same principle, but quite different terminology ;-)
  - Create a **Handle** (in OpenGL also called a **name**)
    ```
    GLint someHandle;
    glGen{Textures|Buffers|...}(1, &someHandle);
    ```
  - Select/activate an object (binding) – a.k.a. *Direct State Access*
    ```
    glBind*(<target>, someHandle);
    ```
  - Pass on data to OpenGL (no need to pass again as long as the data don't change)
  - Deselect an object (unbinding)
    ```
    glBind*(<target>, 0);
    ```
  - When obsolete: delete an object
    ```
    glDelete*(1, &someHandle);
    ```

Vertices in the graphics pipeline

- Primitives are constructed from a stream of vertices
- Which primitives are constructed, is controlled by

  `glDrawArrays(mode, first, count)`

  - e.g., mode = GL_TRIANGLES
- More commands to start the pipeline, e.g., for index-based rendering (no repetitions in data): `glDrawElements(mode, count, type, indices)`



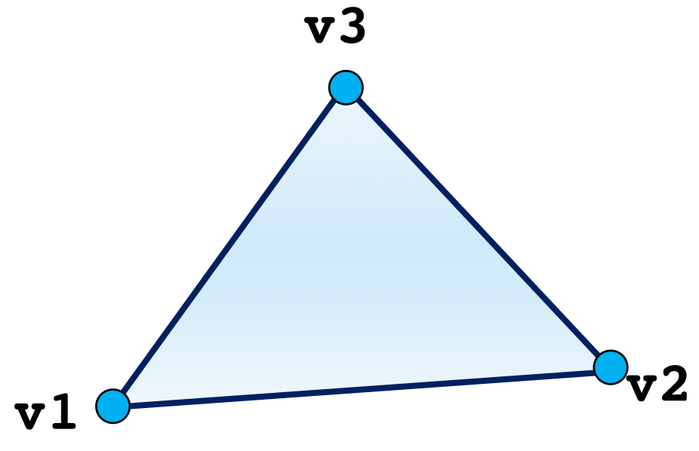
Geometric Primitives

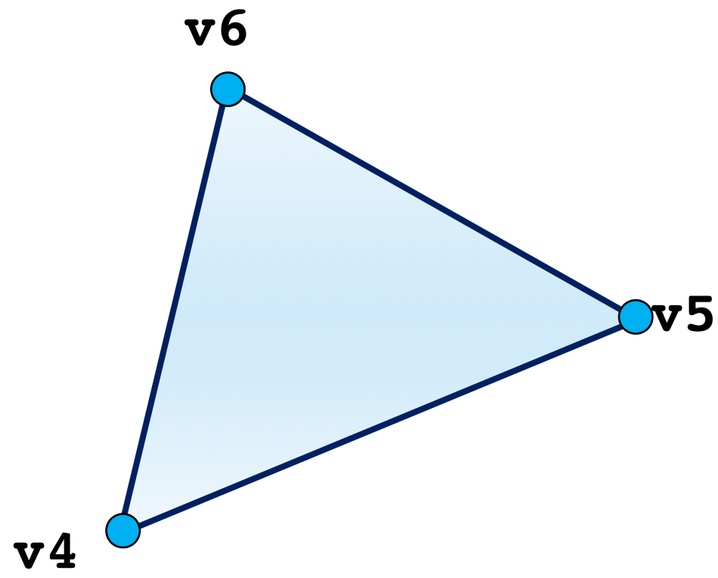
- isolated / independent triangles: **`mode = GL_TRIANGLES`**

```
GLfloat vertices[] = {
v1.x, v1.y, v1.z,
v2.x, v2.y, v2.z,
v3.x, v3.y, v3.z,
v4.x, v4.y, v4.z,
v5.x, v5.y, v5.z,
v6.x, v6.y, v6.z
};
```

Triangle 1

Triangle 2



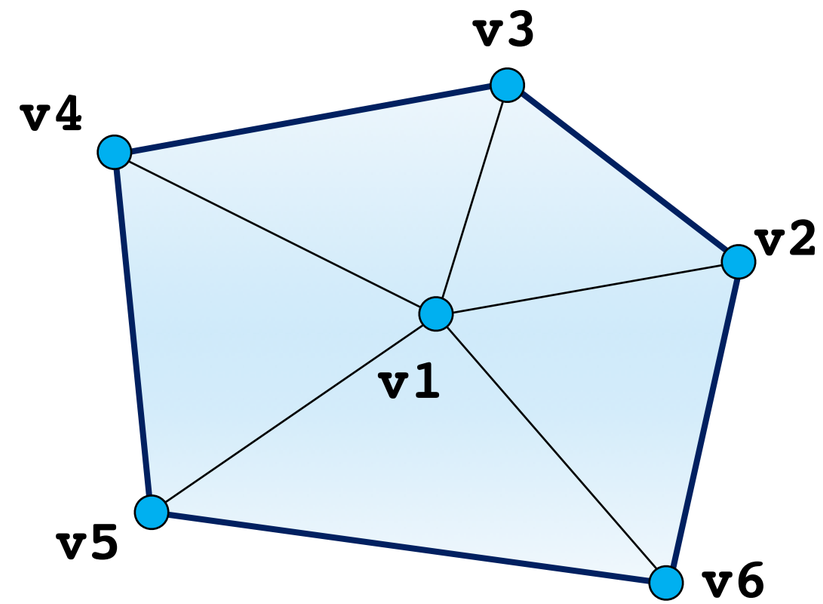


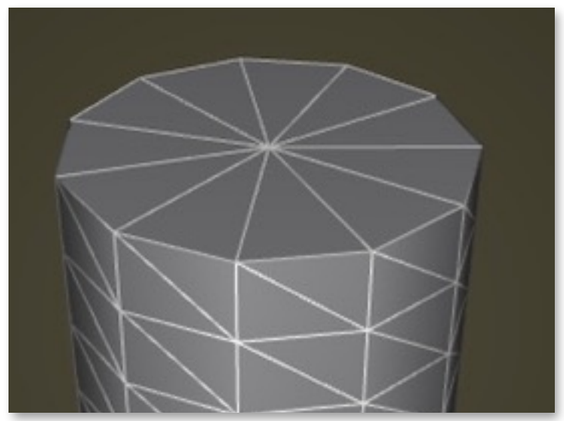
- Vertex-to-triangle-ratio:
  - n triangles
  - 3n vertices processed in geometry-stage (and transferred to GPU)

Geometric Primitives

- Triangle Fan: **`mode = GL_TRIANGLE_FAN`**

```
GLfloat vertices[] = {
v1.x, v1.y, v1.z,
v2.x, v2.y, v2.z,
v3.x, v3.y, v3.z,
v4.x, v4.y, v4.z,
v5.x, v5.y, v5.z,
v6.x, v6.y, v6.z,
v2.x, v2.y, v2.z
};
```

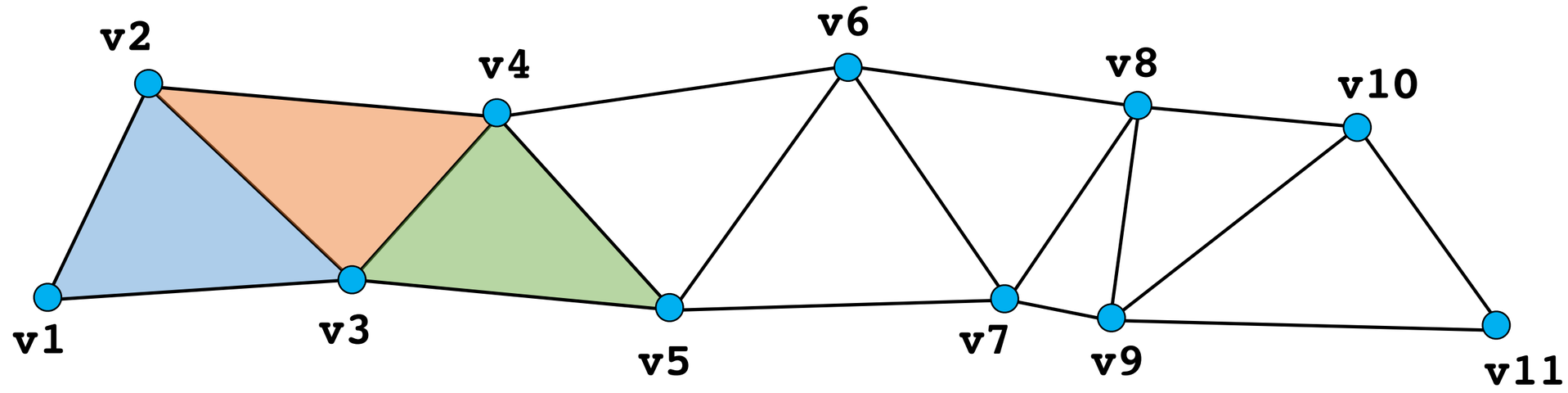




- n triangles
- n + 2 vertices processed
- about one third of workload for geometry-stage (for large n) compared to **`GL_TRIANGLES`**
- but n is typically not really large for triangle fans.

Geometric Primitives

- Triangle Strips: **`mode = GL_TRIANGLE_STRIP`**




```
GLfloat vertices[] = {
v1.x, v1.y, v1.z,
v2.x, v2.y, v2.z,
v3.x, v3.y, v3.z,
v4.x, v4.y, v4.z,
v5.x, v5.y, v5.z,
v6.x, v6.y, v6.z
…};
```

Triangle 1

Triangle 2

Triangle 3

- n triangles, n+2 vertices in geometry-stage
- triangle strips are very efficient

Primitives and Vertex Attributes

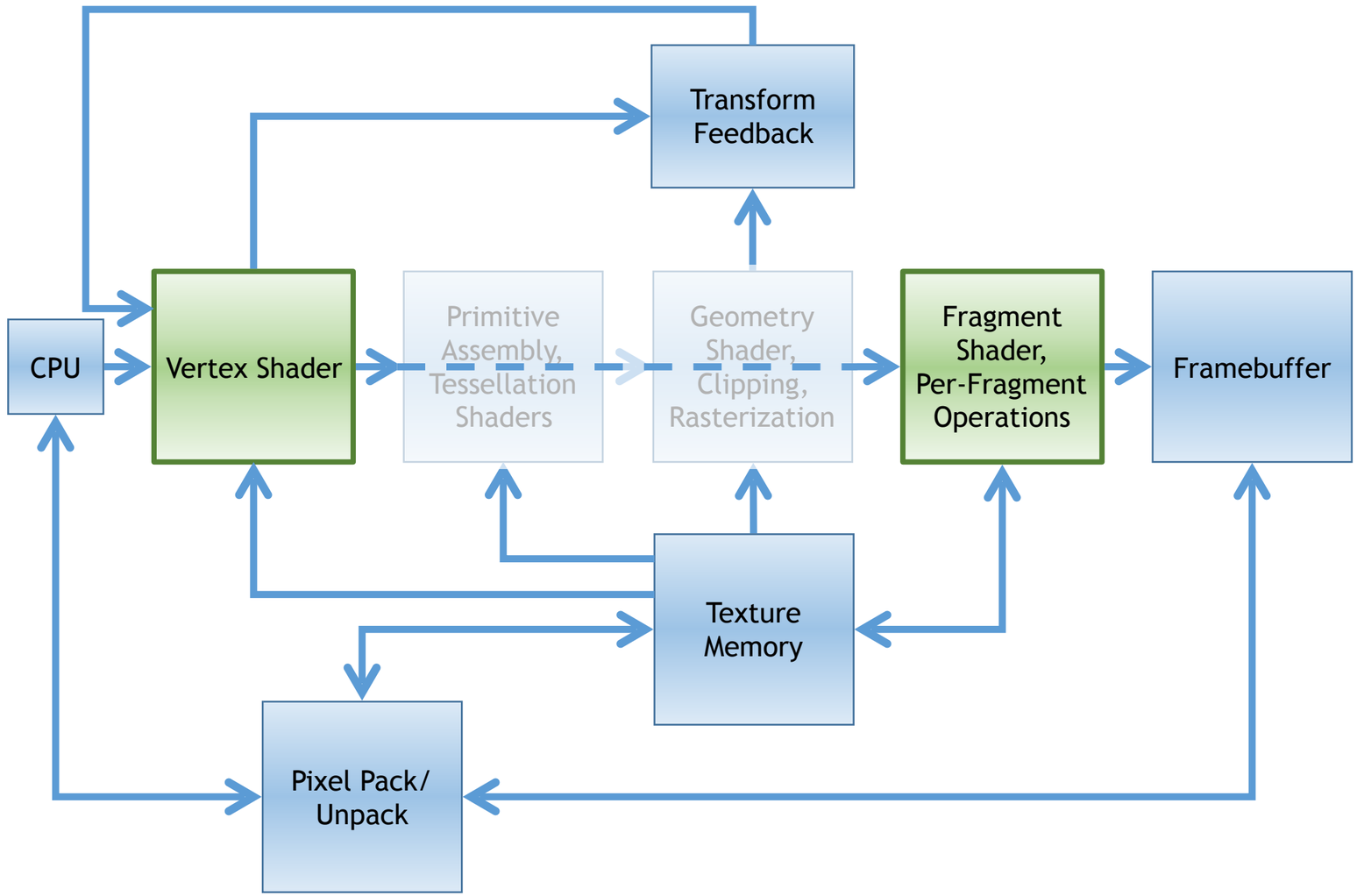
- A vertex has more attributes than just position
  (Nvidia Hardware: often 16 attributes)
- Classical OpenGL: e.g., color, normals, several texture coordinates
- Modern OpenGL allows GL_MAX_VERTEX_ATTRIBS arbitrarily named attributes, e.g.,:
  `glBindAttribLocation(`*`shaderProg`*`, 0, in_position");`
- Access from shader via name
- Remember the state machine:
  - set global states before calling **`glDraw*`**, e.g., **`glPointSize(`*`size`*`);`**

Chapter 4 – OpenGL

- Motivation
- OpenGL Core
- OpenGL Shaders & Shading Language
- OpenGL Objects
- Geometry
- Textures
- Framebuffer
- Tesselation & Geometry Shader

Basic Functionality

- Vertex Shader + Fragment Shader needed for a working geometry and fragment processing
  - more functionality later *(not available in WebGL)*



The smallest possible vertex shader

There is no built-in input attribute!

```
in vec3 in_pos;
uniform mat4 TransformationMatrix;

void main() {
    gl_Position = TransformationMatrix * vec4(in_pos, 1.0);
}
```

Built-in output attribute for vertex shader

- **`void glBindAttribLocation( GLuint program, GLuint index, const GLchar *name )`**
- In Practice:

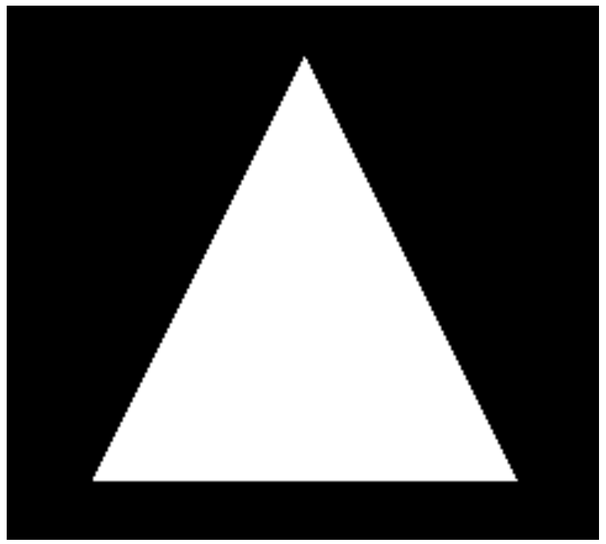
  `glBindAttribLocation(shader, 0, "in_pos")`

The smallest possible fragment shader

There is no Built-In-Out-Attribute! (in OpenGL)

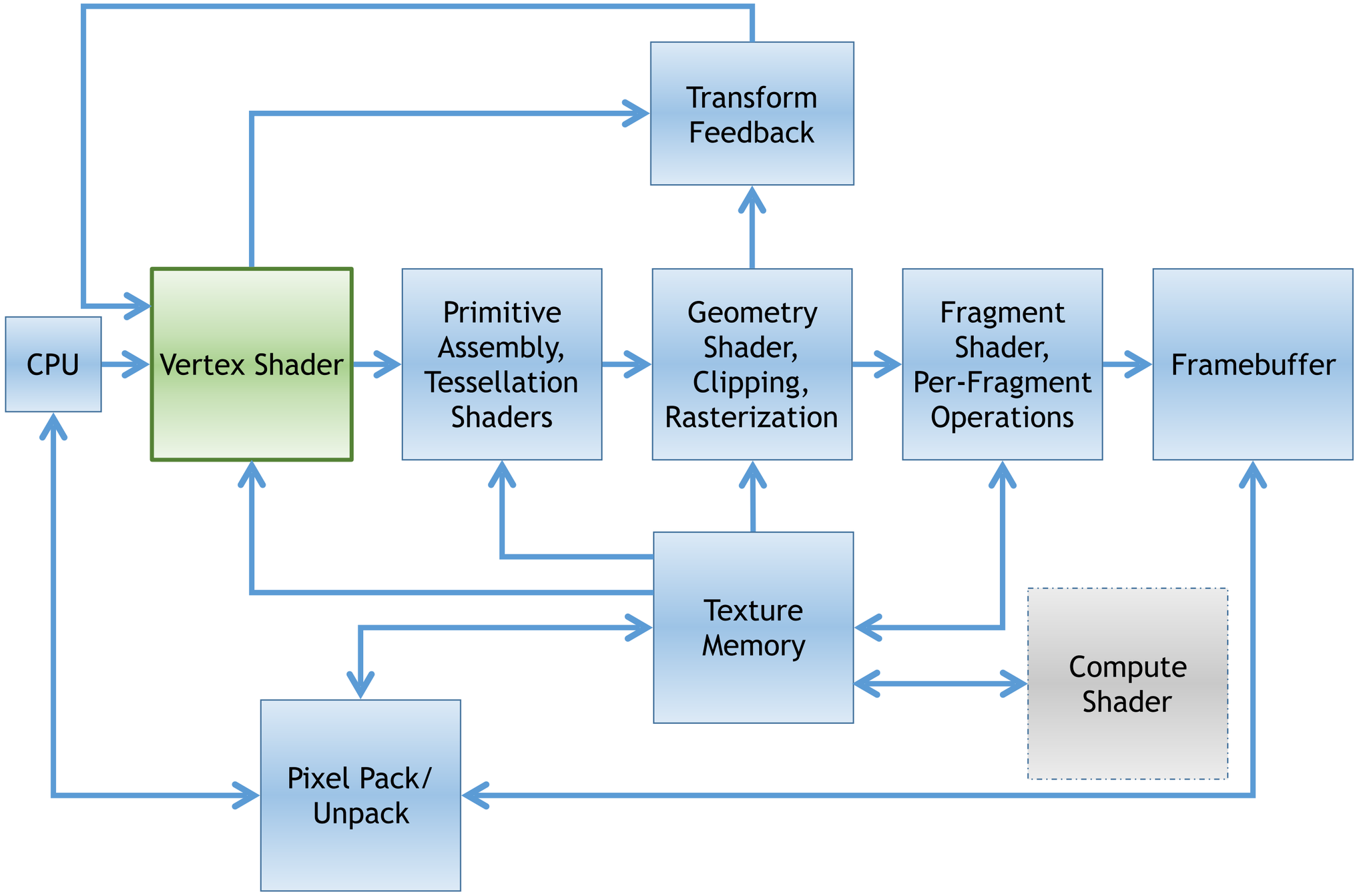
```
out vec4 out_frag_color;

void main() {
    out_frag_color = vec4(1.0, 1.0, 1.0, 1.0);
}
```



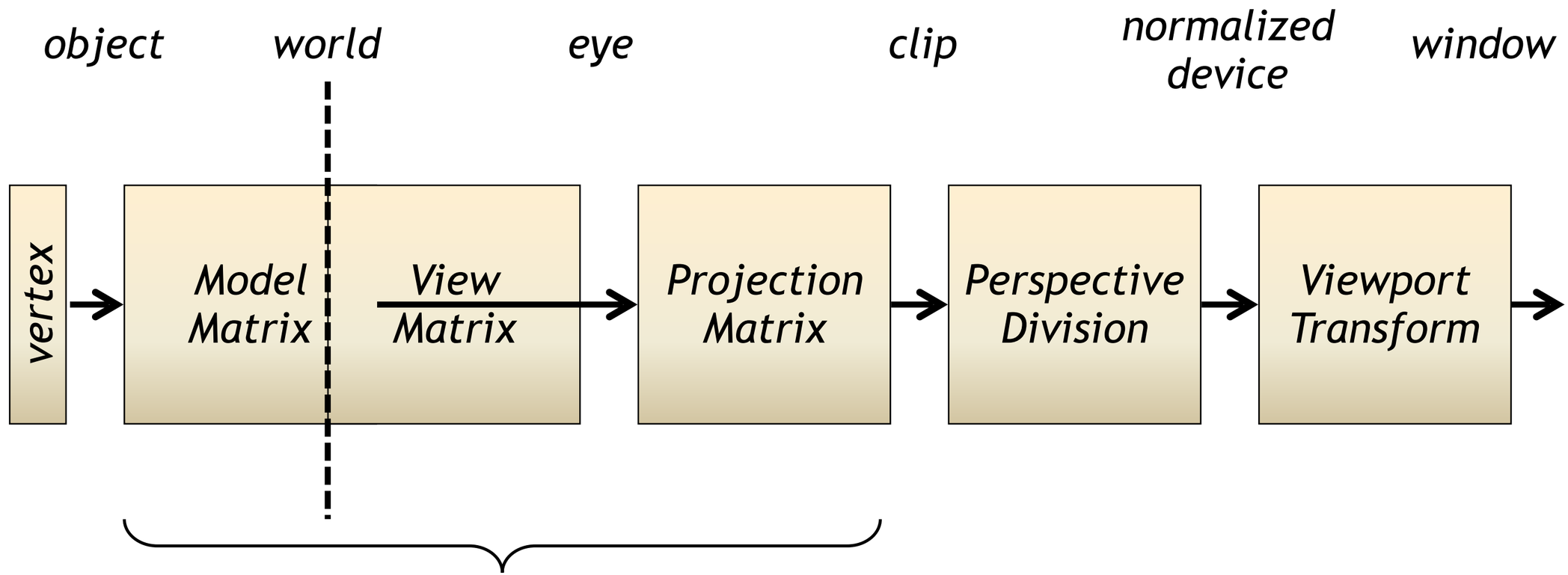
- `void glBindFragDataLocation( Gluint program, GLuint index, const GLchar *name )`
- In Practice:
  `glBindFragDataLocation(shader, 0, "out_frag_color");`

Transformations in the vertex shader



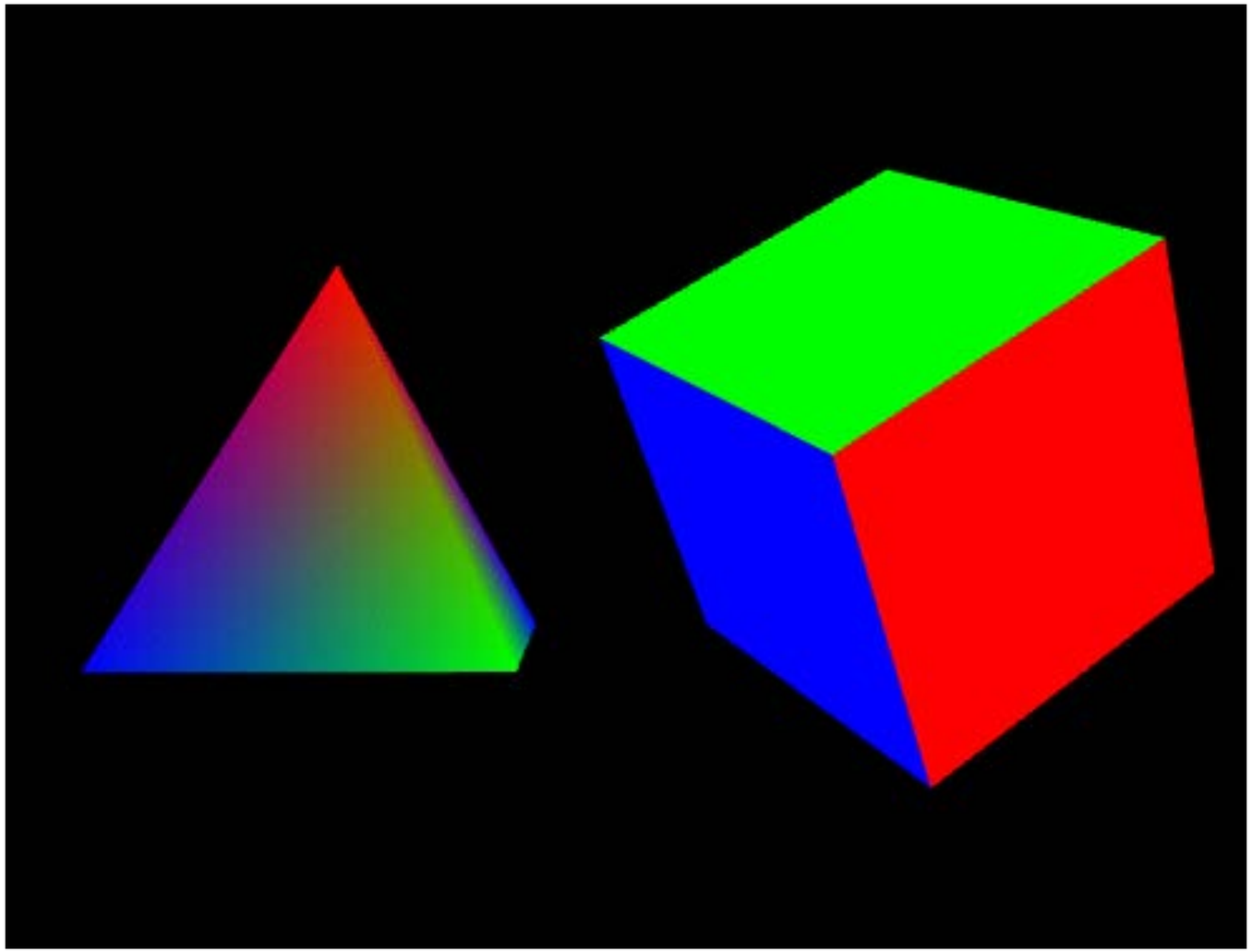
Transformations: Pipeline

- Modern OpenGL lets the programmer deal with matrices (as long as something remains after clipping)
- The concepts from classical OpenGL remain valid



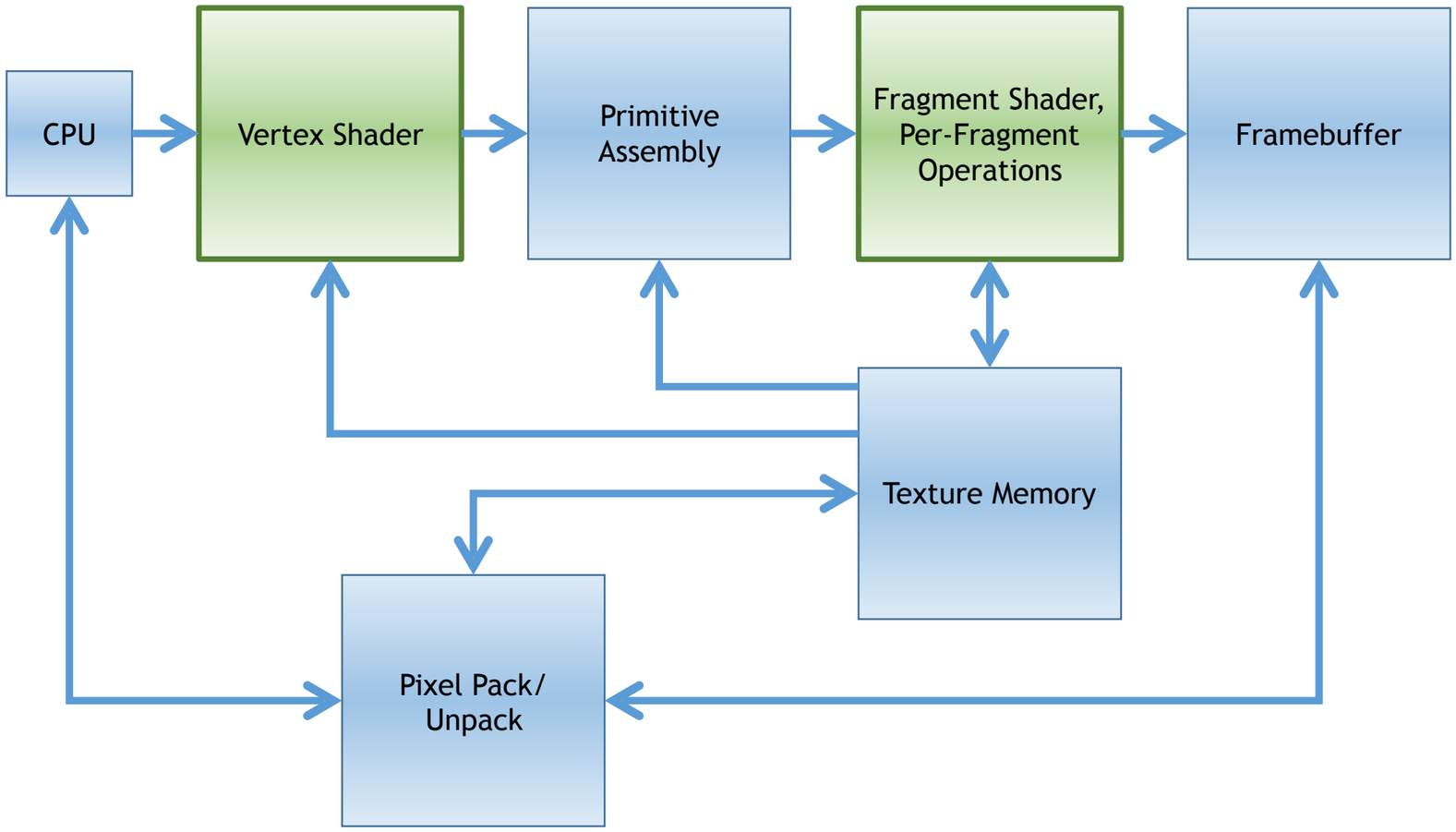
Vertex Attributes

- The color of a vertex can be passed on in a buffer as a vertex attribute
  - depending on rendering technique, there can be more than one
- If it is just forwarded in the pixel shader, it is only interpolated (no shading)



Creating and Using a GLSL Shader Program

- Load shader source code, compile, link, determine attributes, activate, deactivate
- *In Three.js this all is done by the framework*
- *Three.js uses WebGL (1.0 and 2.0)*
  - *No geometry & tessellation shader, no transform feedback (WebGL 1.0)*



Passing on Uniforms

- Uniforms
  - Parameter, which are the same for many/all Vertices/Primitives, are identified via their GLSL variable name (like attributes)
  - Each variable has a „location“ (Index)
    - more efficient than comparing strings each time

```
GLint slot = glGetUniformLocation( GLuint program,
                                   const Glchar *name )
```

  - Setting the values:

```
void glUniform1f( GLint location, GLfloat v0 );
void glUniform{1|2|3|4}f( GLint location, GLfloat ... );
void glUniform1fv( GLint location, GLsizei count,
                   const GLfloat *value );
Void glUniformMatrix4x3fv ( GLint location, GLsizei count,
                GLboolean transpose, const GLfloat *value );
```

  - Which values are the same for all vertices?

Concluding: Triangles

Rasterization, front and backface, backface culling

- Triangles and polygons have a front and a back face
- When seen from the front, the vertices run counterclockwise
- The order can also be explicitly set:

```
glFrontFace( GL_CCW oder GL_CW );
```

- A shader can query the front side with `gl_FrontFacing` (and in turn render differently)
- Triangles seen from the back can be discarded (**Backface Culling**):

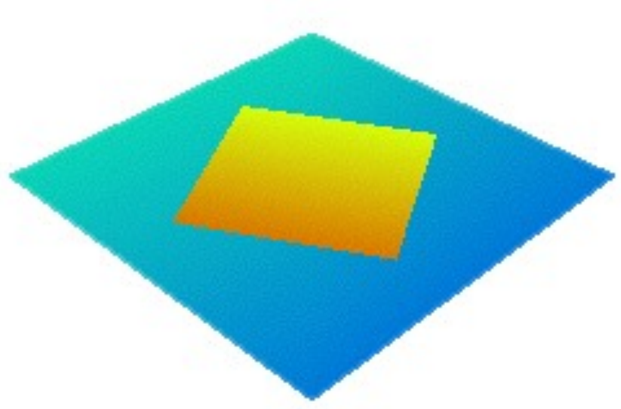
```
glEnable( GL_CULL_FACE );
glCullFace( GL_BACK );
```

  - In closed triangle meshes, the back sides (inside) are never seen.

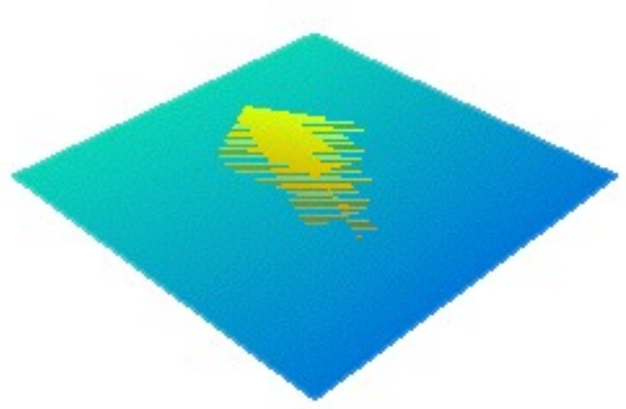
Triangles cont.

- **Z-Fighting**
  - Triangles that overlap in the same plane cause problems
    - e.g when projecting planes and details (surface decals)
    - Rounding erros at depth cause Z-Fighting (Flickering)
    - Depth-Values can be shifted by a constant offset `unit` and a tilt dependent offset `factor` $\cdot \Delta z$
      (positive value causes shift to the back!)

```
glPolygonOffset(factor, units)
glEnable (Gl_POLYGON_OFFSET_{FILL|LINE|POINT})
```



With Offset (Depth-Test „less than or equal“)



Z-Fighting despite Depth-Test „less than or equal“

Optimization in Geometry Processing

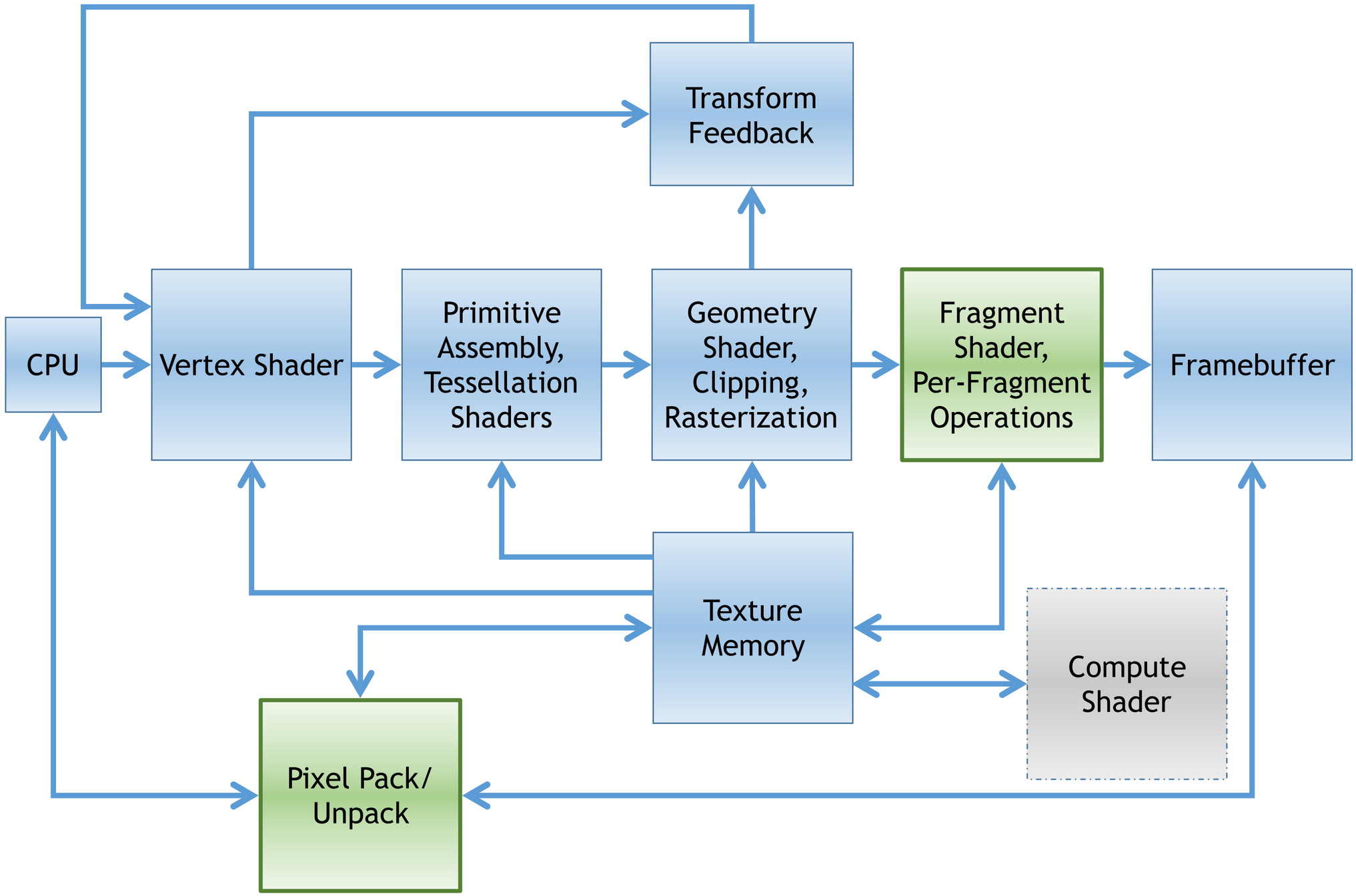
- **Avoid frequent state changes**
  - e.g., by grouping geometry with the same materials or shaders
  - Scene graph APIs do this automatically

- **But: frequently querying the state is *even worse*!**
  - Potential for *Pipeline Stalls*
  - Ergo: keep track of states yourself!

Chapter 4 – OpenGL

- Motivation
- OpenGL Core
- OpenGL Shaders & Shading Language
- OpenGL Objects
- Geometry
- Textures
- Framebuffer
- Tesselation & Geometry Shader

Next Step: Textures in the Fragment Shader



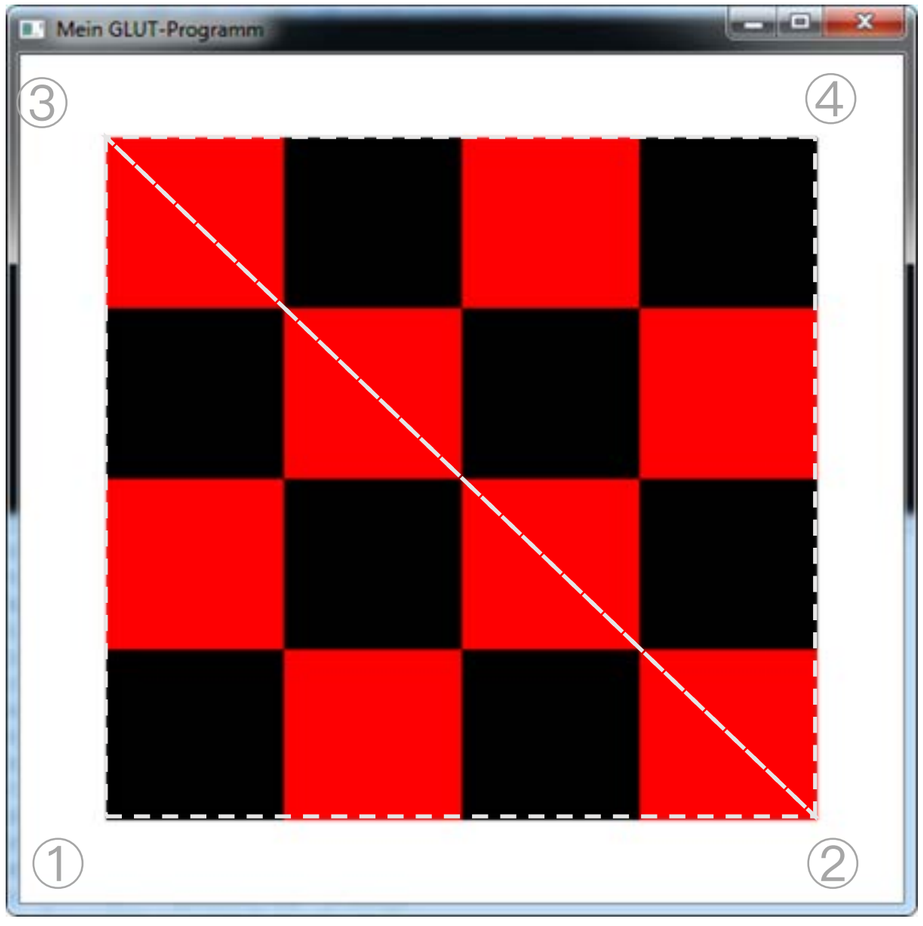
Textures in OpenGL

- Bitmaps with freely selectable properties:
  - Pixel size (up to `GL_MAX_TEXTURE_SIZE;` GeForce GTX1080: 2^{15} = 32768)
  - Number of components (1-4 colors: R to RGBA; plus Depth & Depth+Stencil)
  - Layout (1D, 2D, 3D, Arrays, Cube-Maps etc.)
  - Data type (8, 10, 12bit int per component, 16/32bit float plus many variants, exotic formats (`R11F_G11F_B10F`) and compressed textures)
- Textures are OpenGL objects (like buffers)
- More Information in a *separate chapter* of this class

```
glGenTextures( GLsizei n, GLuint *textures );
glBindTexture( GLenum target, GLuint texture );
glDeleteTextures( GLsizei n, const GLuint *textures );
```

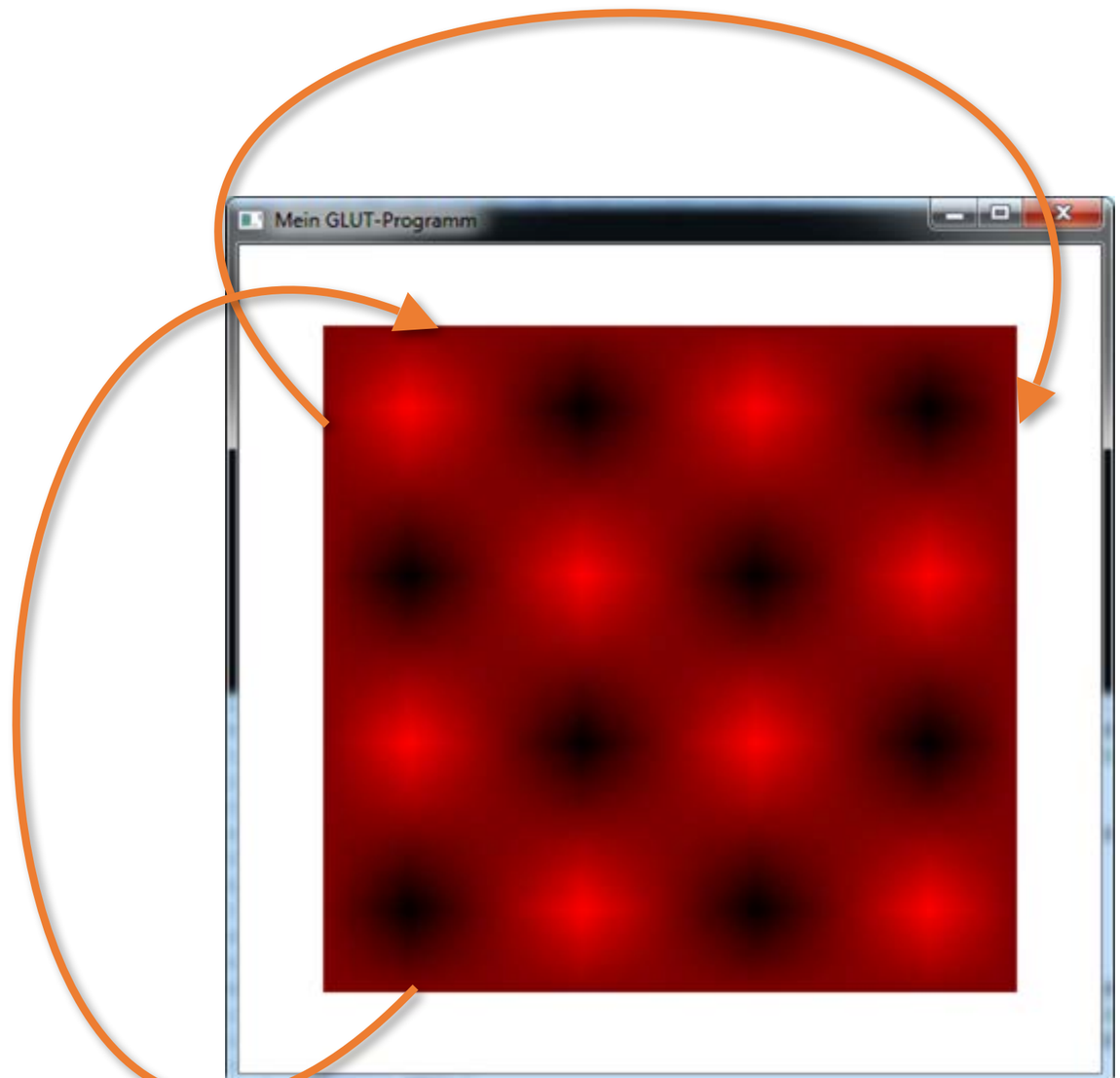
Border Color and Texture coordinates (1)

- Checkerboard texture on a square (= 2 triangles)
-
```
GLfloat quadTexCoords[] = {
     0.f, 0.f, // (1)
     1.f, 0.f, // (2)
     0.f, 1.f, // (3)
     1.f, 1.f, // (4)
};
```
- Sampling: `GL_NEAREST`



Border Color and Texture coordinates (2)

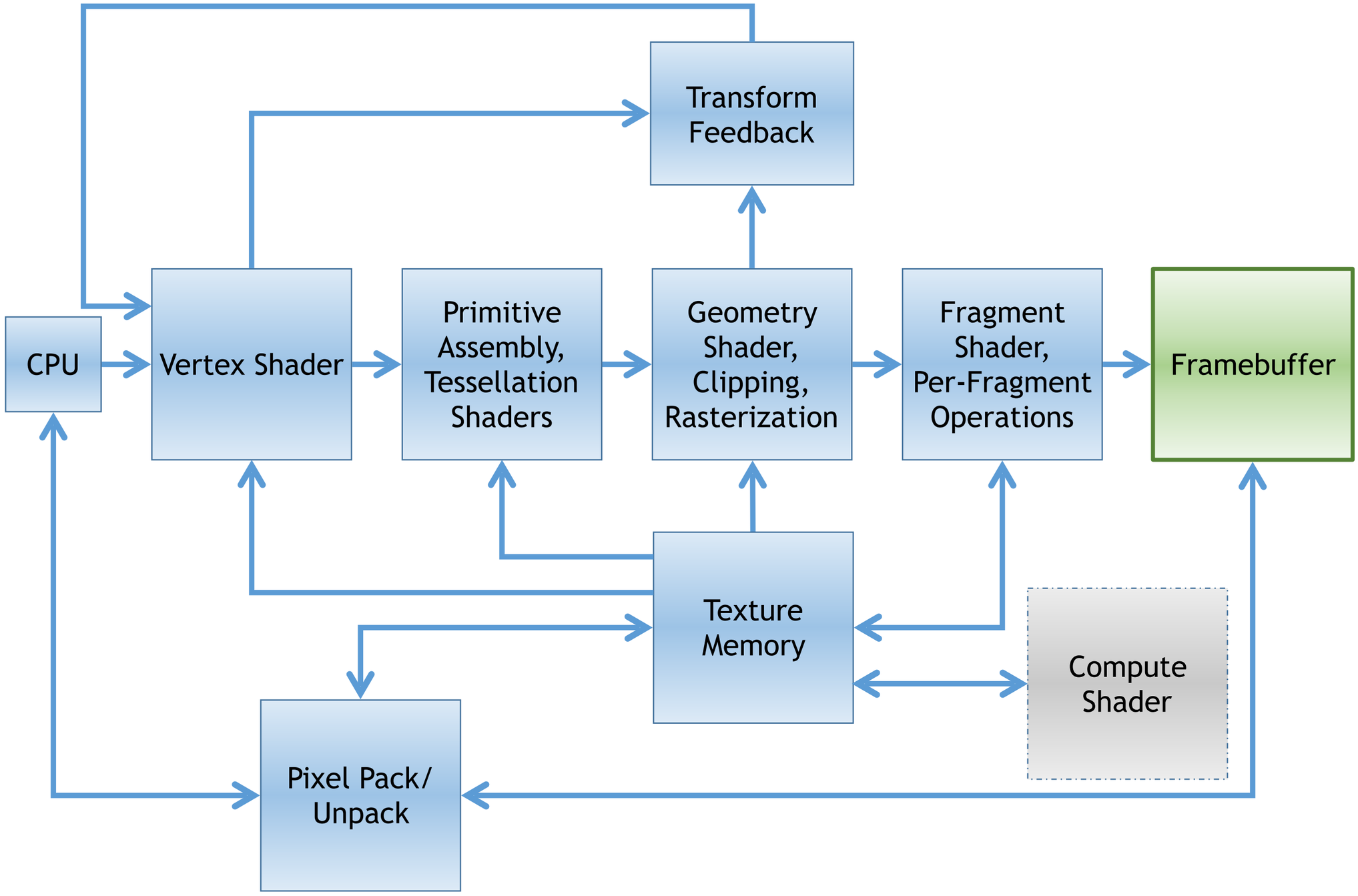
- Checkerboard texture on a square (= 2 triangles)
-
```
GLfloat quadTexCoords[] = {
    0.f, 0.f,
    1.f, 0.f,
    0.f, 1.f,
    1.f, 1.f,
};
```
- Sampling: `GL_LINEAR` (default)
- Wrapping: `GL_REPEAT` (default)
- Periodic Interpolation (possibly unwanted)



Chapter 4 – OpenGL

- Motivation
- OpenGL Core
- OpenGL Shaders & Shading Language
- OpenGL Objects
- Geometry
- Textures
- Framebuffer
- Tesselation & Geometry Shader

Next Step: Framebuffer



Frame Buffer

- A frame buffer consists of
  - a color buffer and a depth/z buffer, but also
  - **a stencil buffer**: often used to store a state per pixel (often 8 bit per pixel)

  → *depth and stencil buffer are optional!*
- *R*equesting the correct frame buffer format:

```
glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE |
                     GLUT_DEPTH | GLUT_STENCIL )
```

- Switching on a depth test against the z-Buffer

```
glEnable( GL_DEPTH_TEST )
```

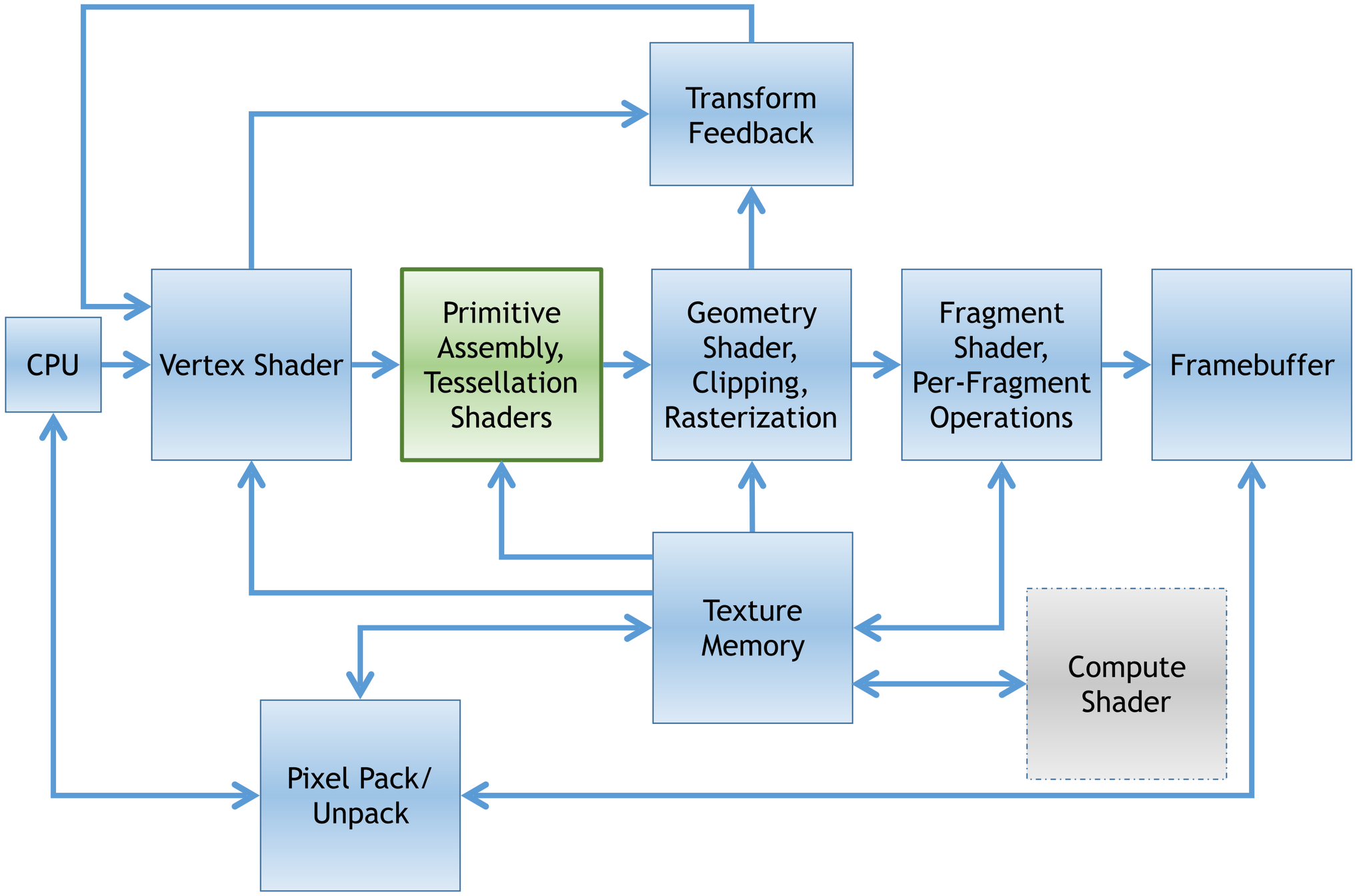
- Clearing (used to be expensive, today always done before drawing!)

```
glClearColor( 0.0, 0.0, 0.0, 1.0);
glClearDepth( 1.0);
glClear( GL_COLOR_BUFFER_BIT |
         GL_DEPTH_BUFFER_BIT |
         GL_STENCIL_BUFFER_BIT )
```

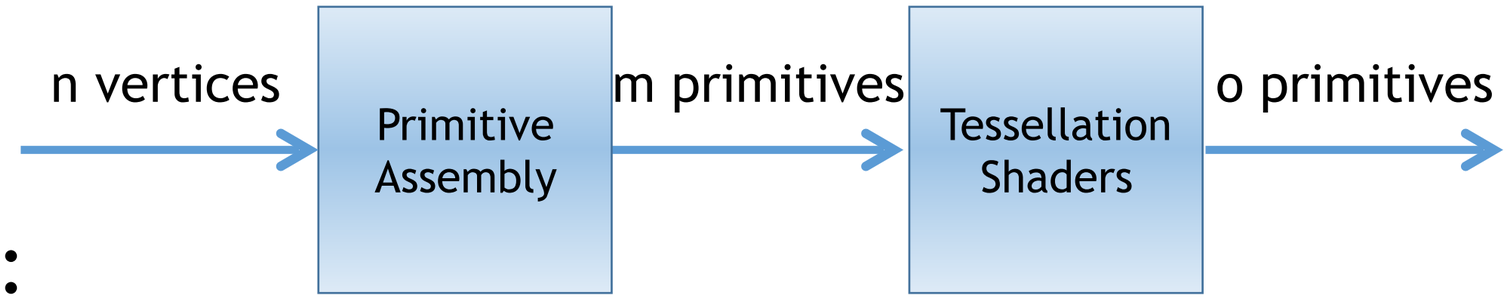
Chapter 4 – OpenGL

- Motivation
- OpenGL Core
- OpenGL Shaders & Shading Language
- OpenGL Objects
- Geometry
- Textures
- Framebuffer
- Tesselation & Geometry Shader

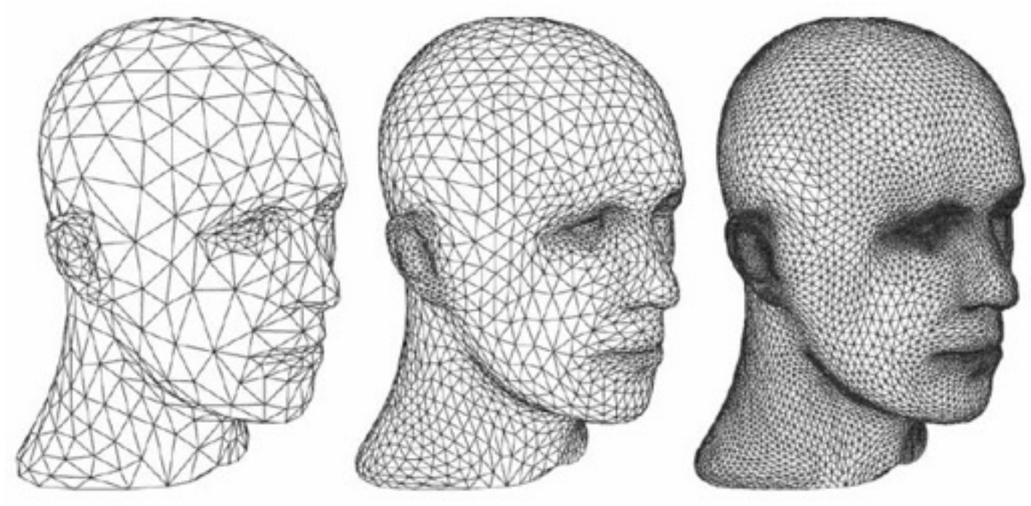
Tessellation



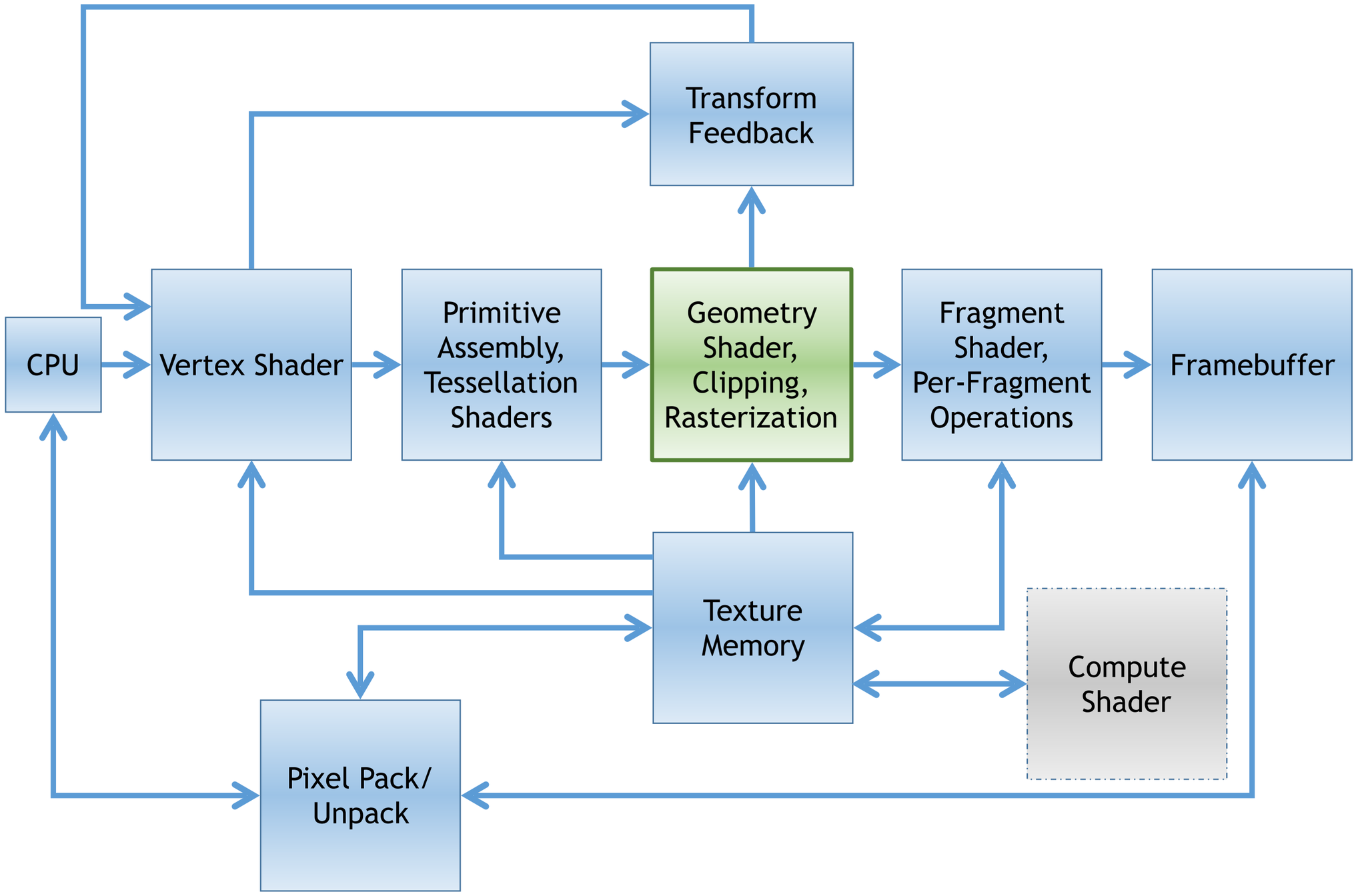
Geometry Processing: Assembly + Tessellation



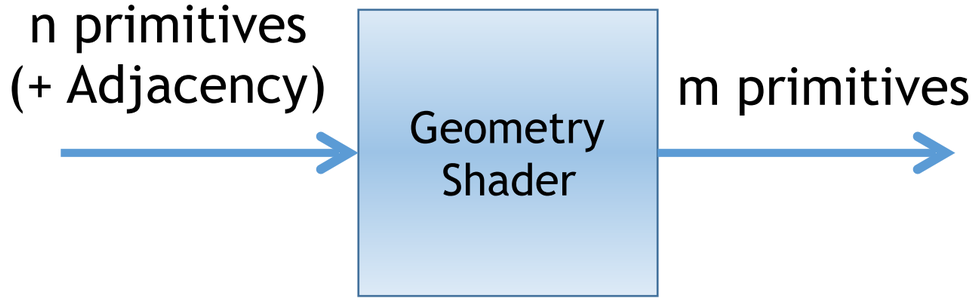
- Tessellation:
  - processes PATCHES from the (simplified) Primitive Assembly
    - All vertices are processed in parallel!
  - **Tessellation Control Shader** determines the subdivision
  - **Tessellator** executes it (can only be parameterized)
  - **Tessellation Evaluation Shader** works on the resulting geometry



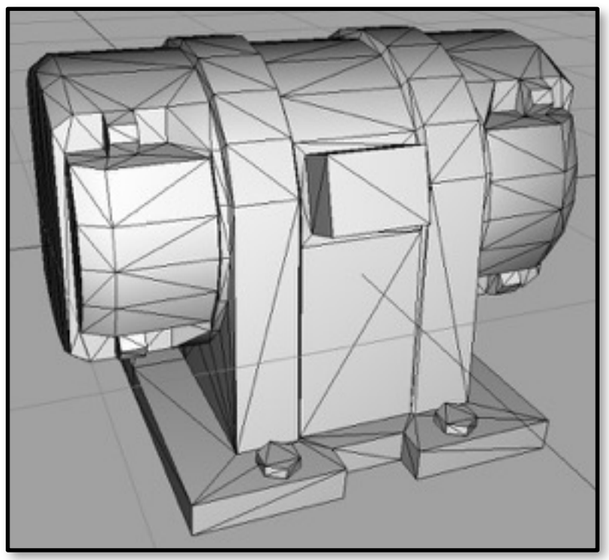
Geometry Shader

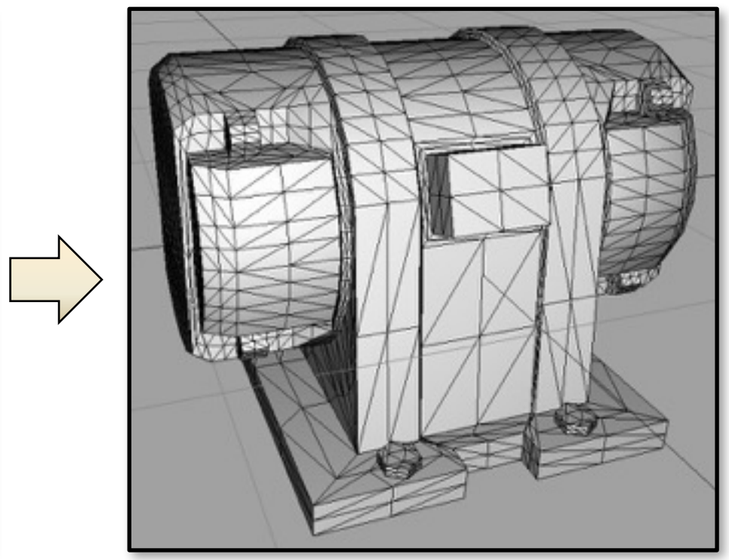


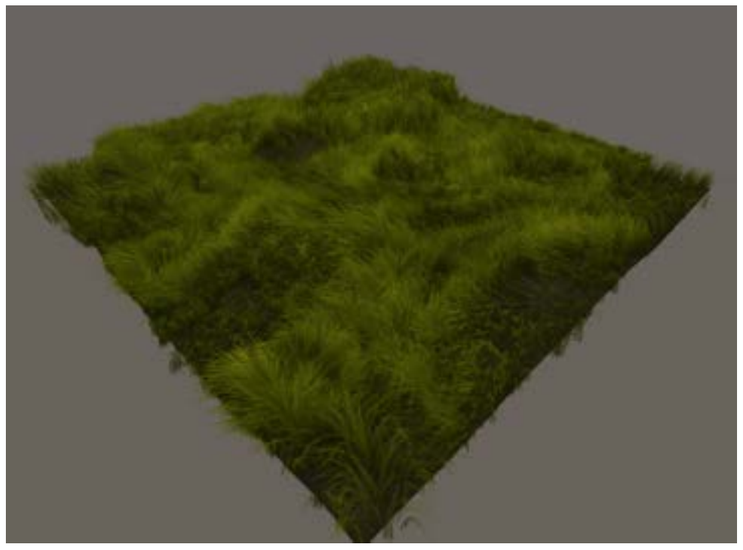
Geometry Processing: Geometry Shader



- Geometry Shaders
  - Work on arbitrary primitives sequentially per primitive
  - Extend primitives - but only output limited attributes per primitive
    `MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS` (GeForce GTX 500: 1024)
  - Delete or modify primitives (vertices to triangles etc.)







reddit: u/kekeciam