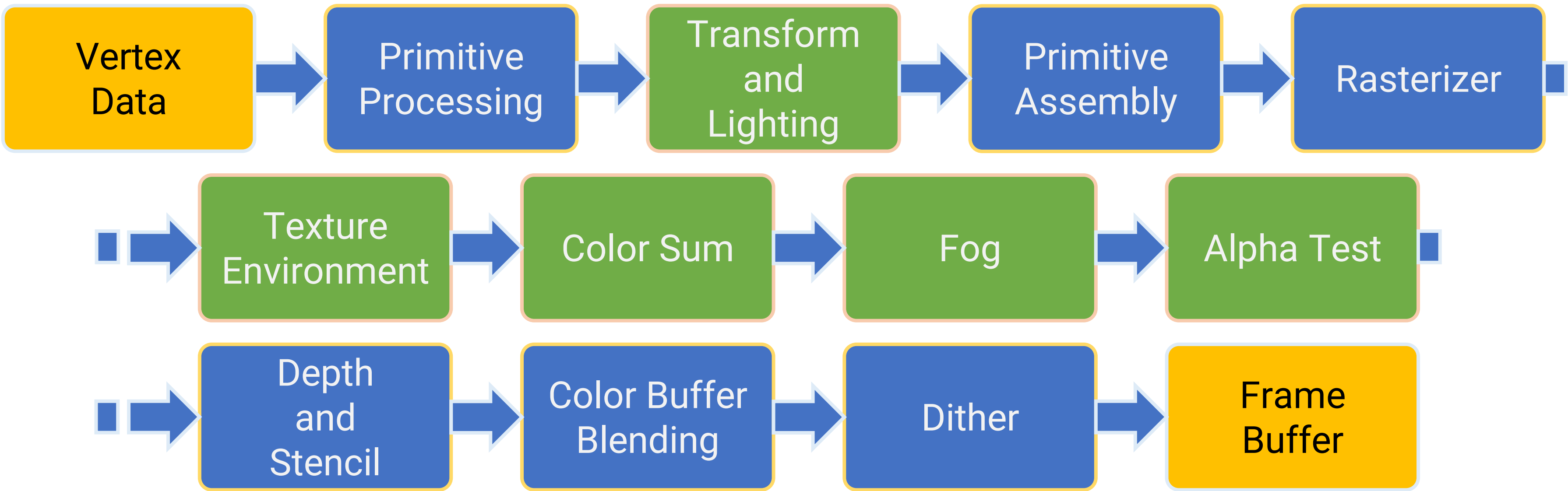
Advanced Shaders

Computer Graphics

Yu-Ting Wu

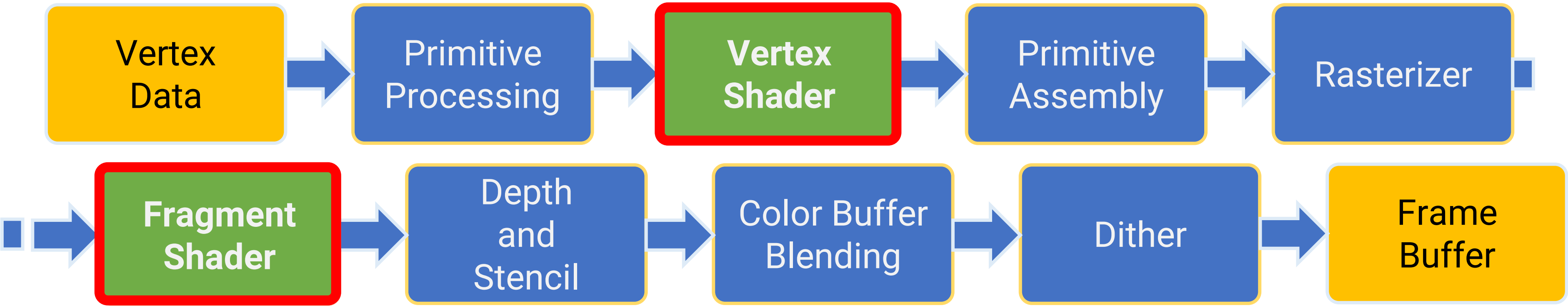
Recap: OpenGL 1.1 (Fixed Function Pipeline)

- Used when OpenGL was first introduced
- All the functions performed by OpenGL are **fixed** and **could not** be modified except through the manipulation of the **rendering states**



Recap: OpenGL 2.0

- Introduce **Vertex** and **Fragment** shaders to replace some fixed stages for providing more flexibility



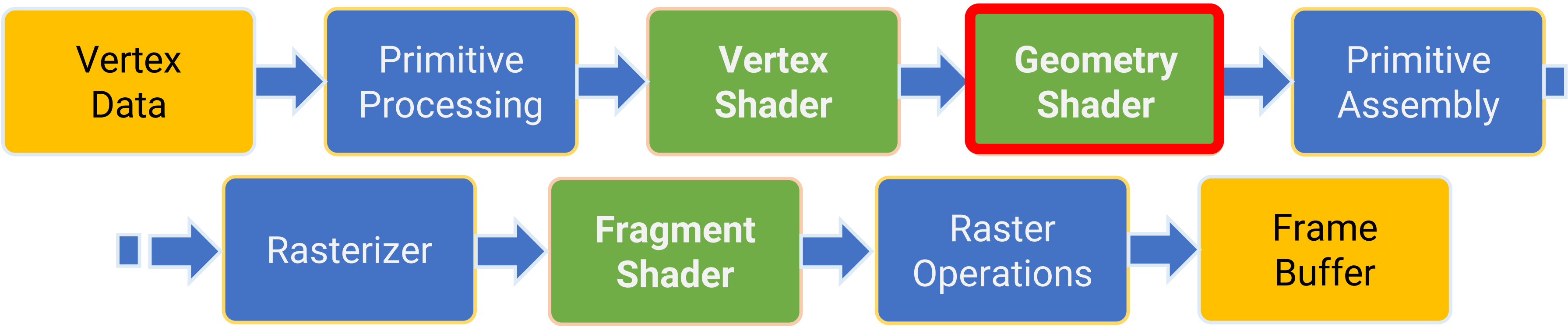
Important Shader Timeline

- OpenGL 1.0 (1992): fixed function pipeline
- OpenGL 2.0 (2004): vertex/fragment shader
- OpenGL 3.2 (2009): geometry shader
- OpenGL 4.0 (2010): tessellation shader
- OpenGL 4.3 (2012): compute shader

Geometry Shader

OpenGL 3.2: Geometry Shader

- Vertex shader processes each vertex separately
- What if we would like to manipulate a **primitive**, such as a **line** or a **triangle**?
- For this reason, ***OpenGL 3.2*** adds **Geometry** shader for **per-primitive processing**

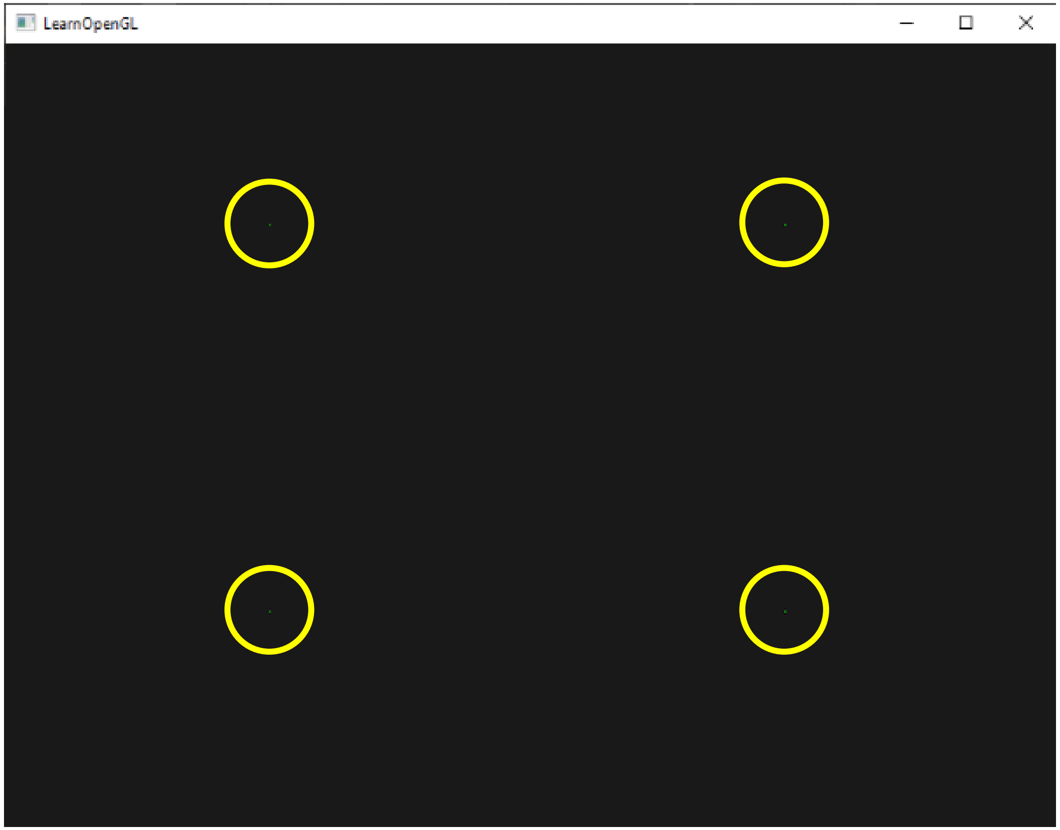


Geometry Shader

- An **optional** stage
- Take a set of vertices that form a **single primitive** as input, such as
  - Points
  - Lines
  - Triangles
- A geometry shader can transform the primitives with **different transforms for each vertex** or
- **Generate new primitives (on GPU)**

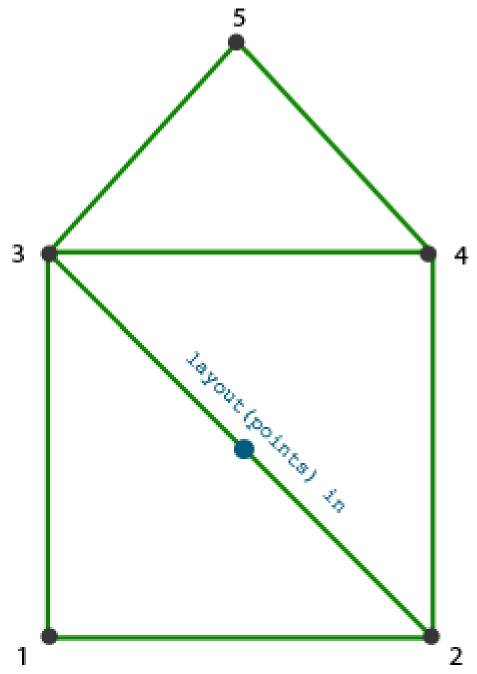
Geometry Shader (cont.)

- An example for the overall picture
  - Input primitive streams: 4 points



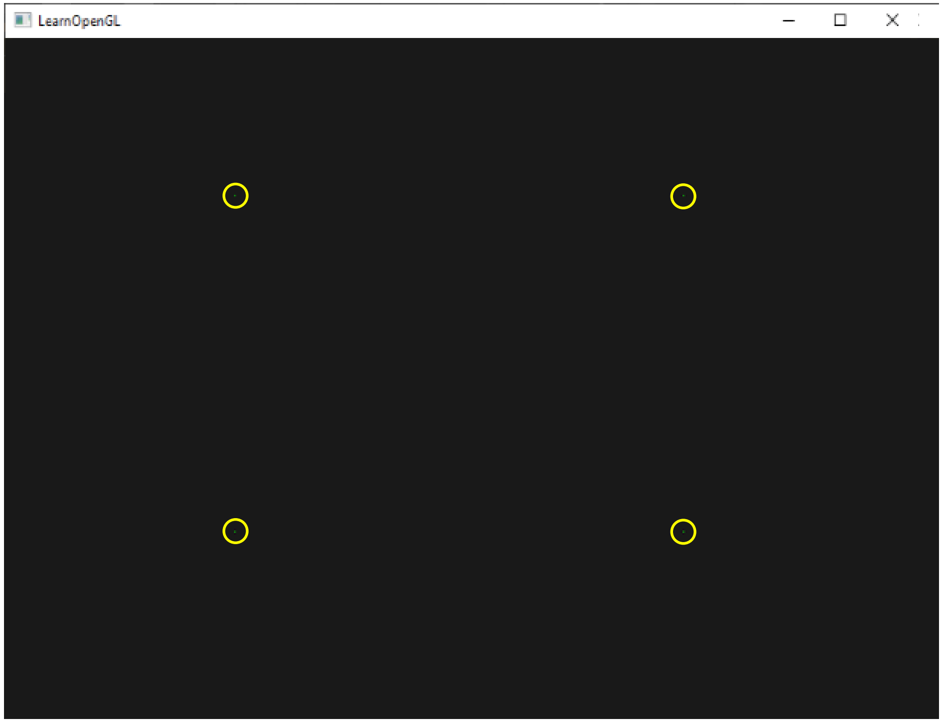
Geometry Shader (cont.)

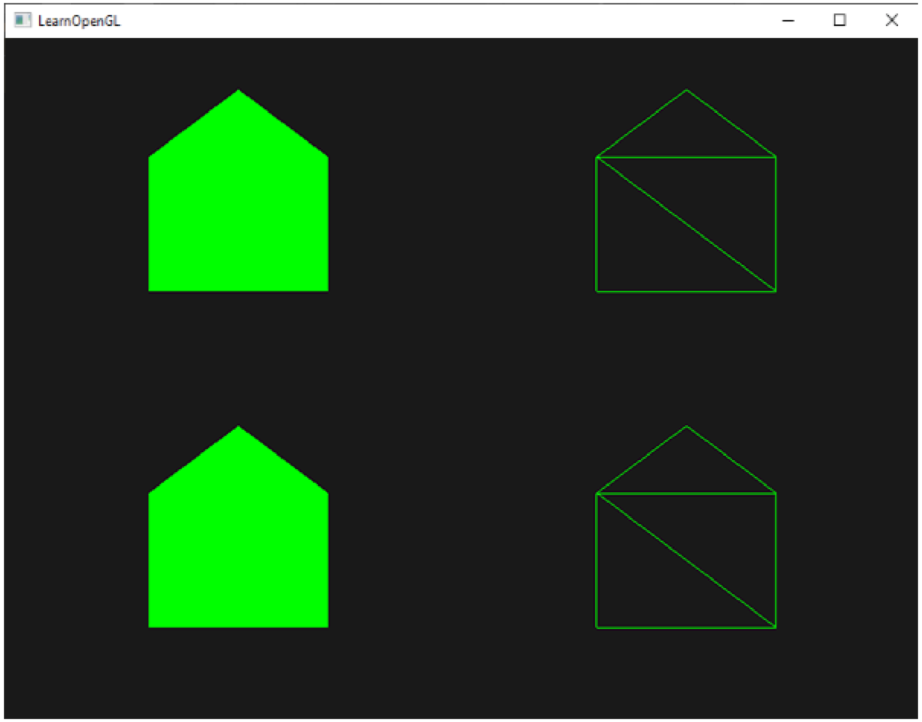
- An example for the overall picture
  - For each primitive (in this case, a point), generate 5 vertices with different offsets



Geometry Shader (cont.)

- An example for the overall picture
  - Output



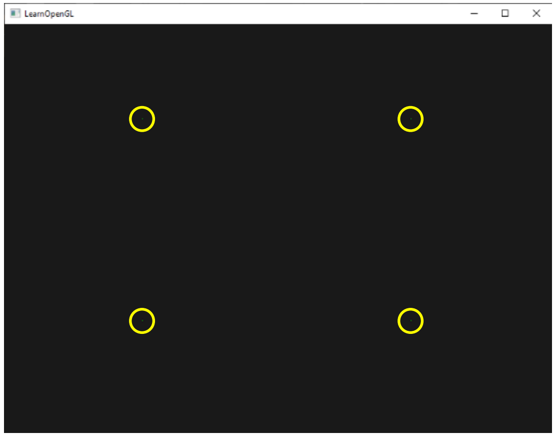


Geometry Shader (cont.)

- Code snippet
  - **Vertex data**

```
float points[] = {
        -0.5f,  0.5f, // top-left
         0.5f,  0.5f, // top-right
         0.5f, -0.5f, // bottom-right
        -0.5f, -0.5f  // bottom-left
};
```

```
glDrawArrays(GL_POINTS, 0, 4);
```



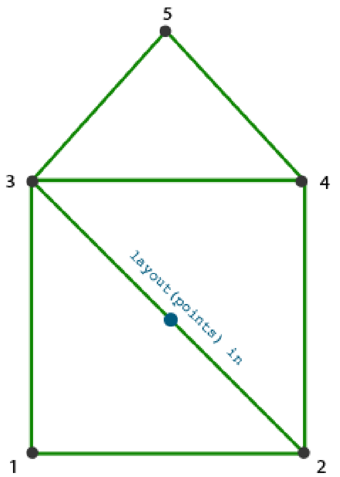
  - **Vertex Shader**

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

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
}
```

Geometry Shader (cont.)

- Code snippet
  - **Geometry Shader**

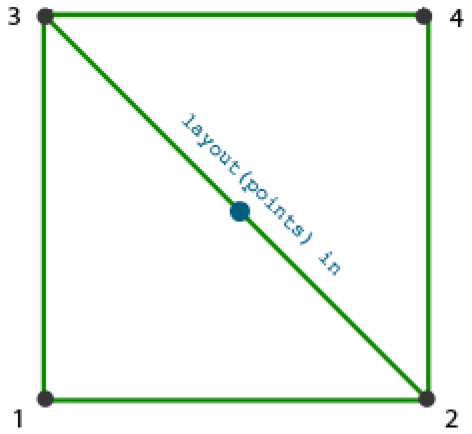
```
#version 330 core
layout (points) in;
layout (triangle_strip, max_vertices = 5) out;

void build_house(vec4 position)
{
    gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0);    // 1:bottom-left
    EmitVertex();
    gl_Position = position + vec4( 0.2, -0.2, 0.0, 0.0);    // 2:bottom-right
    EmitVertex();
    gl_Position = position + vec4(-0.2,  0.2, 0.0, 0.0);    // 3:top-left
    EmitVertex();
    gl_Position = position + vec4( 0.2,  0.2, 0.0, 0.0);    // 4:top-right
    EmitVertex();
    gl_Position = position + vec4( 0.0,  0.4, 0.0, 0.0);    // 5:top
    EmitVertex();
    EndPrimitive();
}

void main() {
    build_house(gl_in[0].gl_Position);
}
```



Applications: Particle System

- https://youtu.be/tUAAltGNTaI

(0, 1) (1, 1)

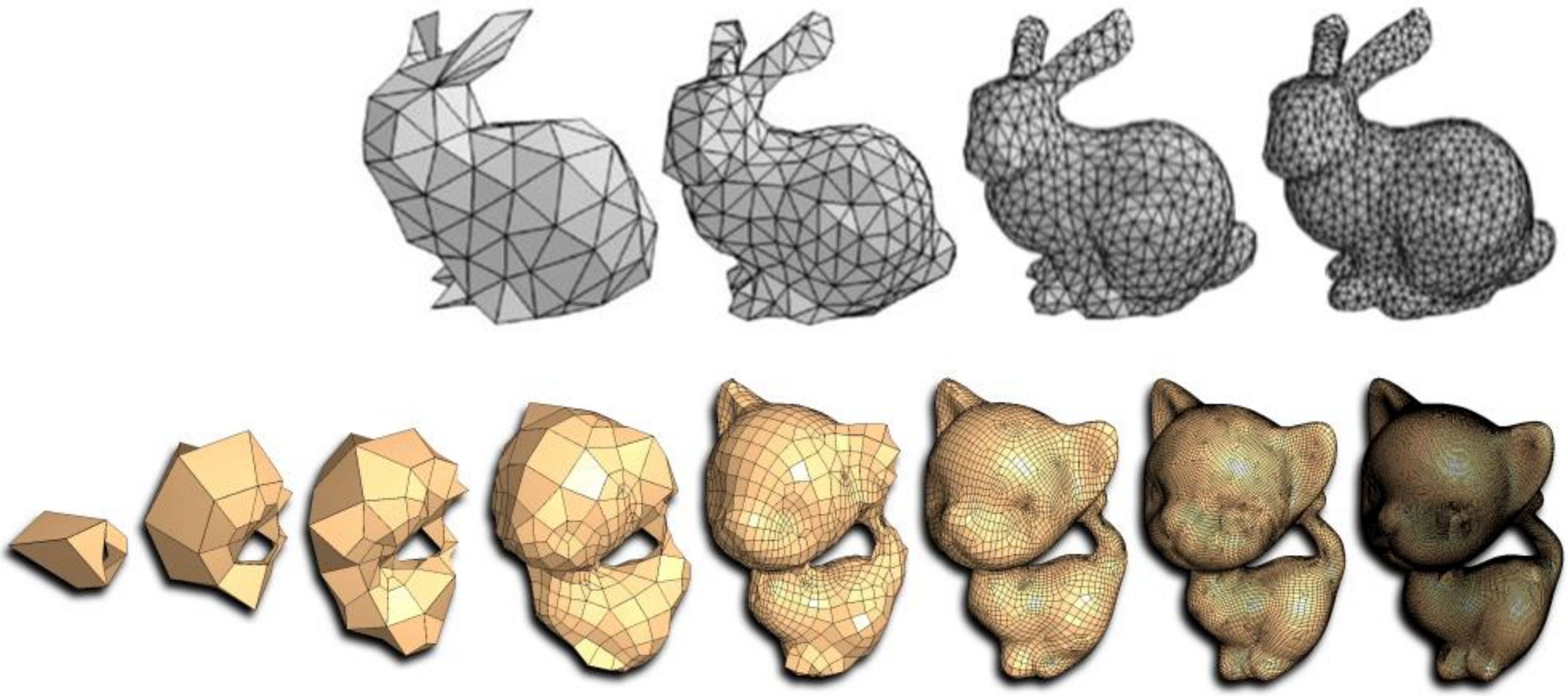


(0, 0) (1, 0)

Tessellation

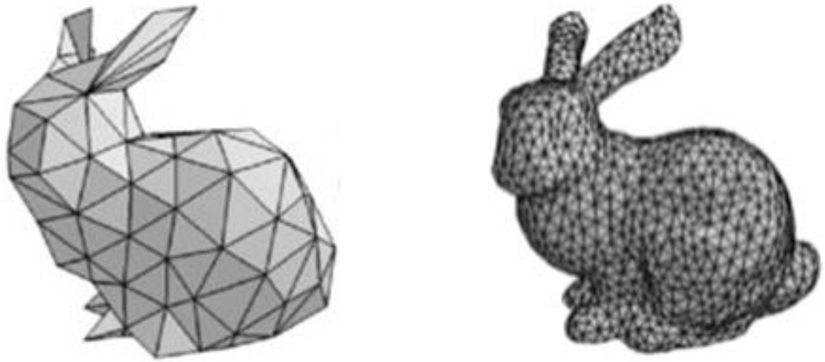
Background

- Recall that using more triangles can lead to higher-quality meshes; however, at the expense of taking more time to render



Background (cont.)

- When we look at a complex model up close, we prefer to use a highly-detailed model



- When we look at it from a great distance, we prefer to use a rough one because it only projects to a few pixels

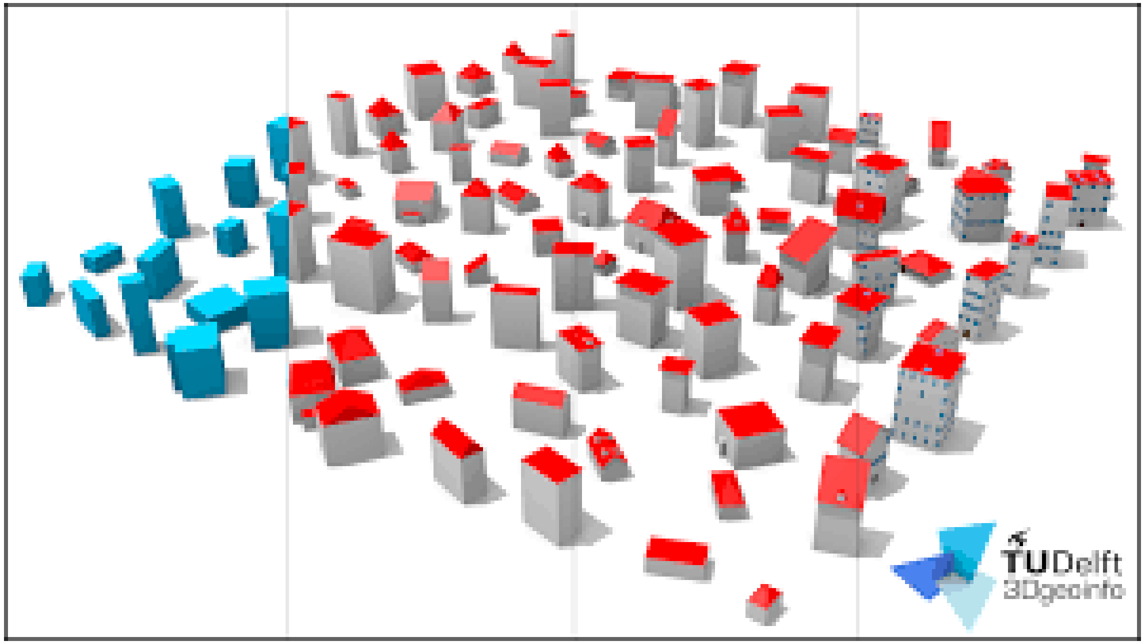
- One solution to this problem is using **Levels of Detail (LOD)**

Background (cont.)

- **Level of Details (LOD)**
  - Artists create the same model at multiple levels of detail

Background (cont.)

- **Level of Details (LOD)**
  - Artists create the same model at multiple levels of detail
  - We can then select the version to use based on some criterion, such as the distance from the camera

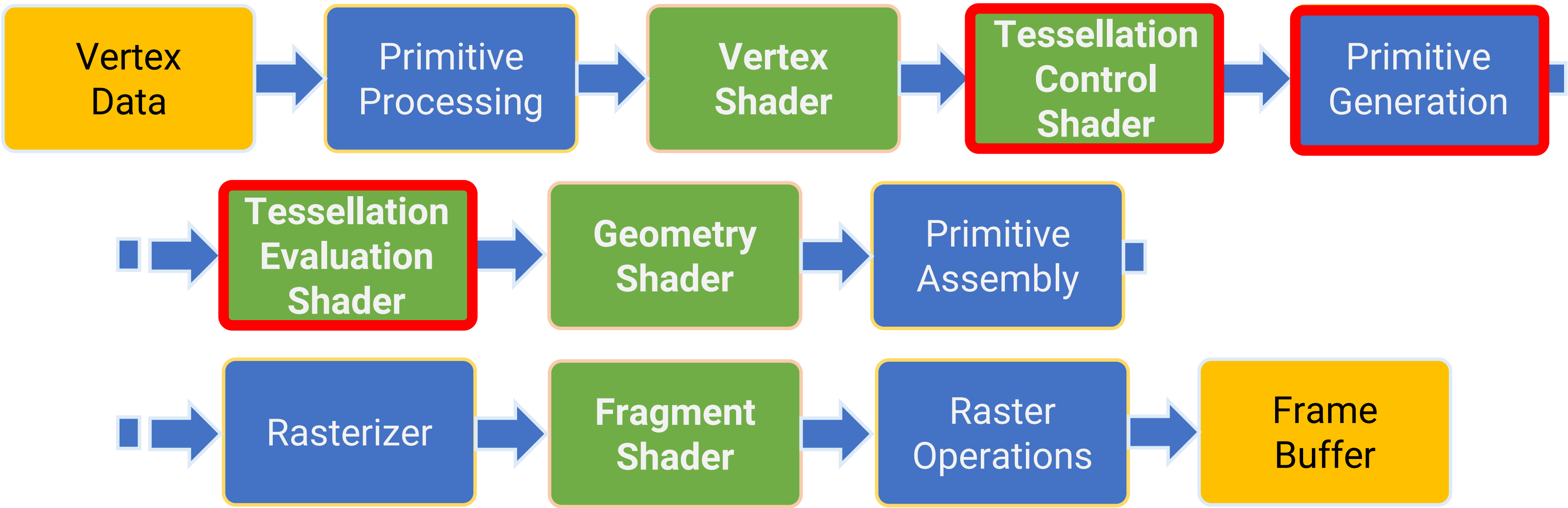


Background (cont.)

- **Level of Details (LOD)**
  - Artists create the same model at multiple levels of detail
  - We can then select the version to use based on some criterion, such as the distance from the camera
  - **However, this requires more artist resources, and the level of models might dynamically change over time**
    - **The change of LOD should also be smooth!**
  - Can we start with a low polygon model and subdivide each triangle on the fly into smaller triangles? The answer is **tessellation**!

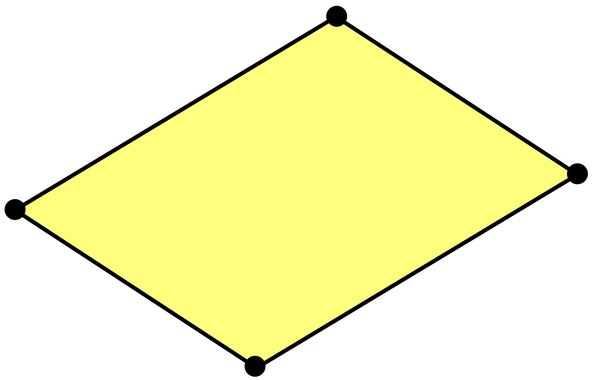
OpenGL 4.0: Tessellation

- OpenGL 4.0 adds tessellation into the graphics pipeline
- It comprises **two new shaders**, **Tessellation Control Shader (TCS)** and **Tessellation Evaluation Shader (TES)**, and **a fixed stage**, **Primitive Generation**

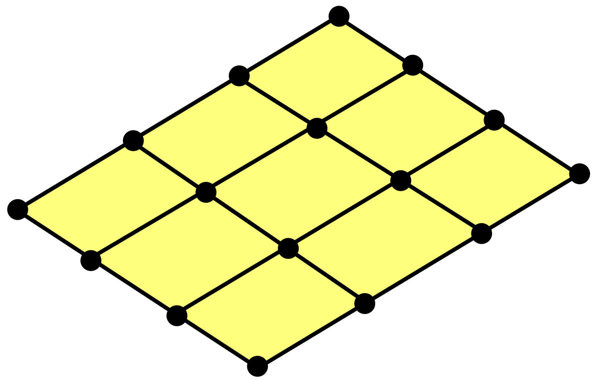


OpenGL Tessellation (cont.)

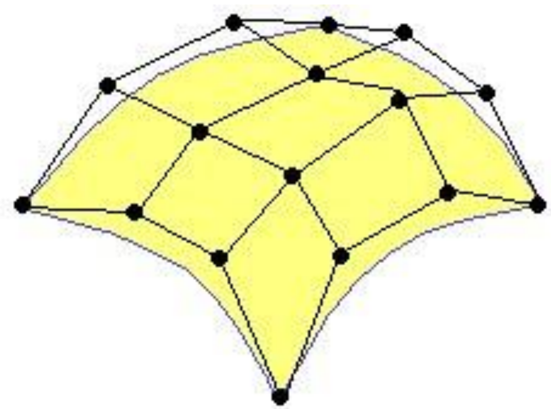
- Overview



original patch
(line, triangle)



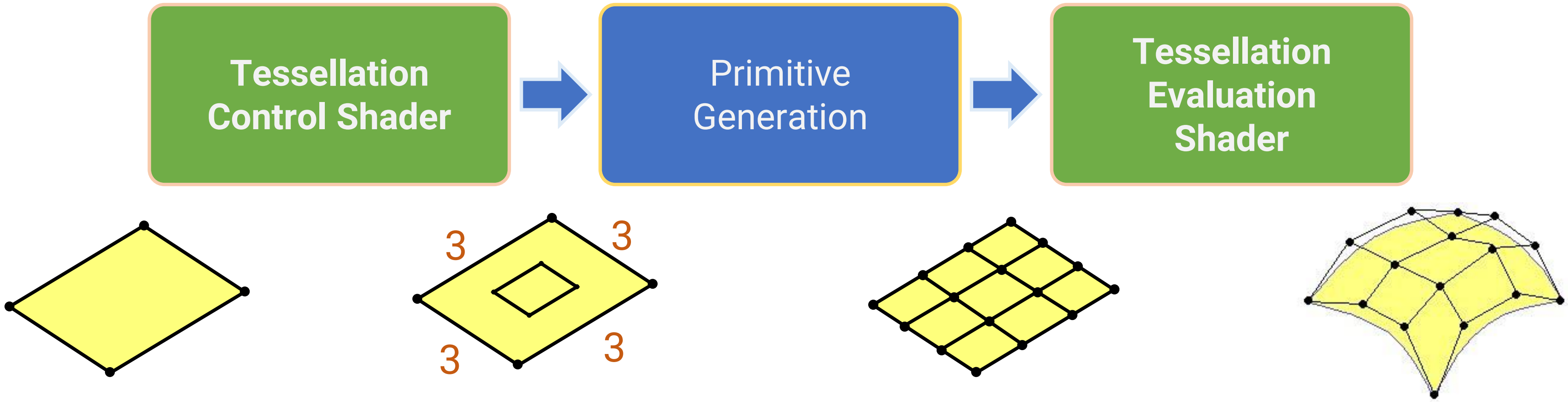
subdivided



adjust vertex positions
based on some formulas
(e.g., Bezier curve)

OpenGL Tessellation (cont.)

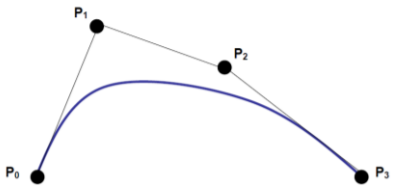
- Overview



determine tessellation levels

hardware interpolation

evaluate vertex position



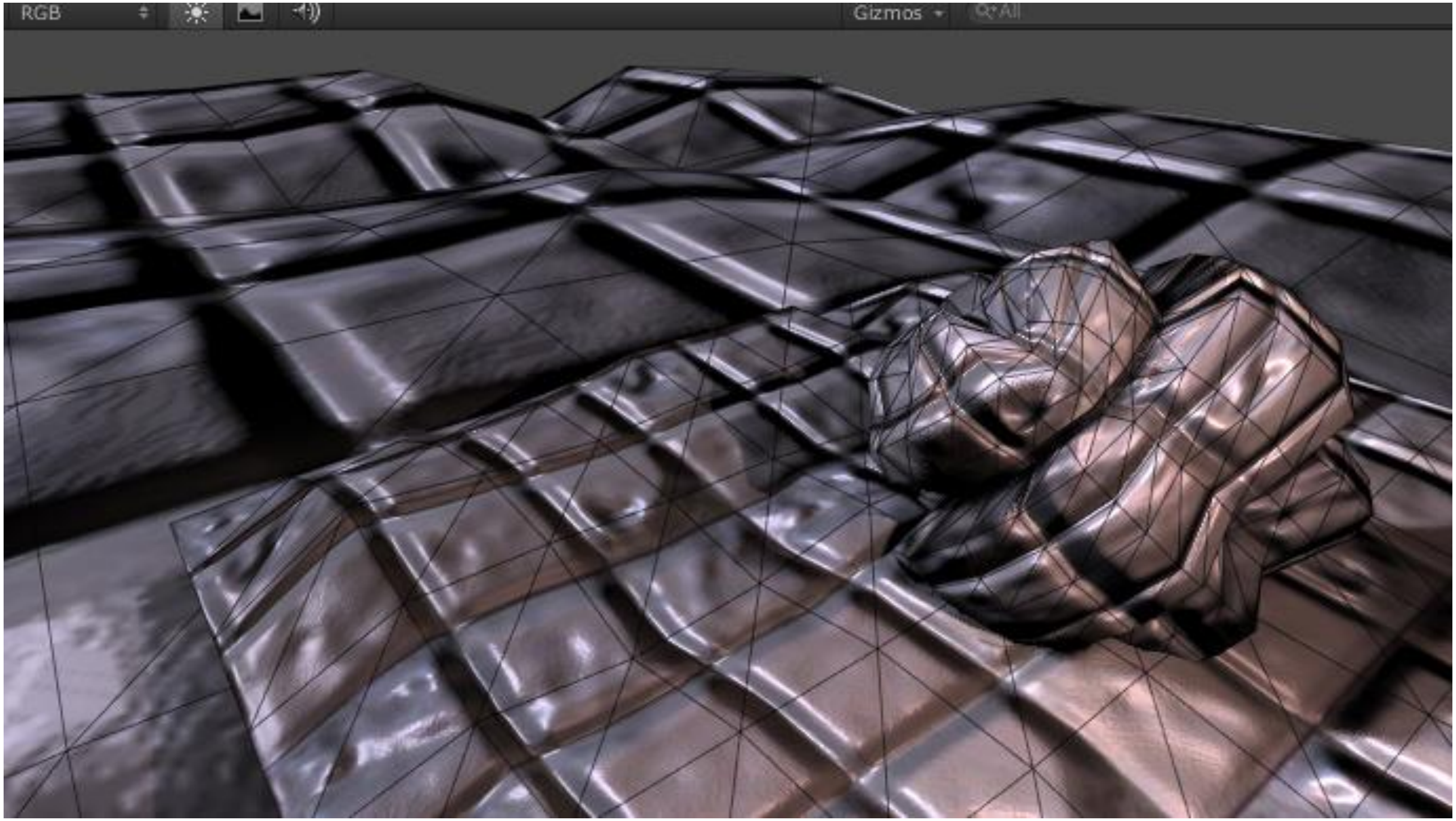
$$P(u) = (1-u)^3 P_0 + 3u(1-u)^2 P_1 + 3u^2(1-u)P_2 + u^3 P_3$$

OpenGL Tessellation (cont.)

- **Advantages**
  - Send less vertex/index data from CPU to GPU (save bandwidth)
  - More flexible (and smooth) level-of-details (LOD)

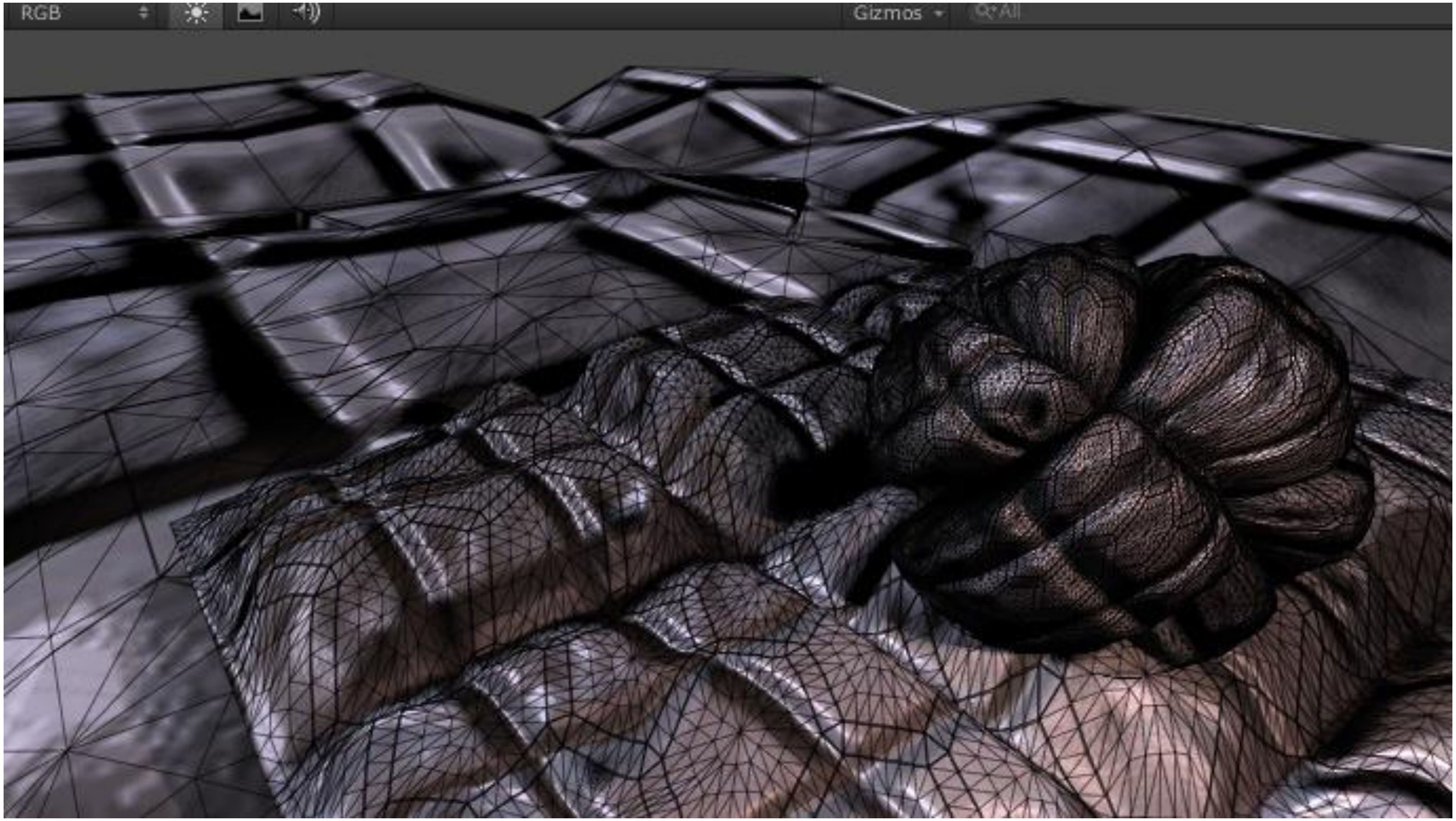
OpenGL Tessellation (cont.)

- An example result: no tessellation



OpenGL Tessellation (cont.)

- An example result: with tessellation shader



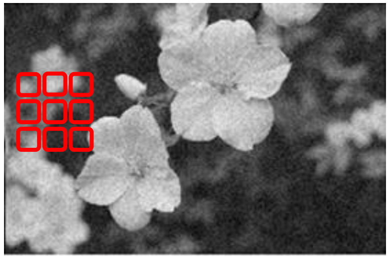
Compute Shader

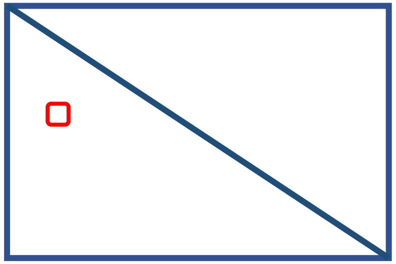
Background

- Traditionally the graphics card (GPU) has been a rendering co-processor which is handling graphics
- It got more and more common to use graphics cards for other (not necessarily graphics-related) computational tasks, called **General Purpose Computing on Graphics Processing Units (GPGPU)**
  - Higher parallelism
  - Faster floating-point calculation
- In OpenGL 4.3, **Compute Shaders** are introduced for computing arbitrary information

Advantages of GPGPU

- Before GPGPU (including CUDA, OpenCL, and Compute Shader), if you want to use GPU for performance improvement, you need to **translate the target problem into a rendering problem**
- For example, to filter an image, you need to
  - Draw a quad (two triangles) into a frame buffer object
  - Bind the input image as a texture
  - Lookup the texture and perform filtering in the fragment shader





Compute Shader v.s. Other Shaders

- Compute shaders are **NOT** part of the graphics pipeline
- It uses a **function (kernel)** to run over a set of the input data (stream) and output a set of data (stream), without any assumptions of the data types and format
  - **You can consider the vertex/fragment shaders as kernels with fixed data types (vertex/fragment data)**
- Each element is processed independently in parallel
- Directly make changes on the GPU memory, similar to a **void** function

A Simple Compute Shader

```
layout(local_size_x = 16, local_size_y = 16) in;
layout(rgba8, binding = 0) uniform restrict readonly image2D u_input_image;
layout(rgba8, binding = 1) uniform restrict writeonly image2D u_output_image;

const int M = 16;
const int N = 2 * M + 1;

// sigma = 10
const float coeffs[N] = float[N](...); // generated kernel coefficients

void main() (kernel, executed by each data item)
{
    ivec2 size = imageSize(u_input_image);
    ivec2 pixel_coord = ivec2(gl_GlobalInvocationID.xy);

    if (pixel_coord.x < size.x && pixel_coord.y < size.y)
    {
        vec4 sum = vec4(0.0);

        for (int i = 0; i < N; ++i)
        {
            for (int j = 0; j < N; ++j)
            {
                ivec2 pc = pixel_coord + ivec2(i - M, j - M);
                if (pc.x < 0) pc.x = 0;
                if (pc.y < 0) pc.y = 0;
                if (pc.x >= size.x) pc.x = size.x - 1;
                if (pc.y >= size.y) pc.y = size.y - 1;

                sum += coeffs[i] * coeffs[j] * imageLoad(u_input_image, pc);
            }
        }

        imageStore(u_output_image, pixel_coord, sum);
    }
}
```

Compute Shader v.s. CUDA & OpenCL

- There are more popular GPGPU APIs like **NVIDIA CUDA** and **OpenCL** offer more features as they are aimed at heavyweight GPGPU projects
- The **OpenGL Compute Shader** is intentionally designed to incorporate other OpenGL functionality and uses **GLSL** to make it easier to integrate with the existing OpenGL graphics pipeline/application
- Common applications of Compute Shader
  - Physical simulation
  - Real-time image processing / texture editing
  - Collision detection
  - GPU ray tracer

Summary

- The input and output of the six different shaders in OpenGL

Stage	Data Element
Vertex Shader	per vertex
Tessellation Control Shader	per vertex (in a patch)
Tessellation Evaluation Shader	per vertex (in a patch)
Geometry Shader	per primitive
Fragment Shader	per fragment
Compute Shader	per (abstract) "work item"


END OF PRESENTATION
ANY QUESTIONS?