Computer Graphics - Exercise 12

Maximilian Richter

Winter Semester 2022/23

1 OpenGL and Shaders

OpenGL and GLSL are defined standards with detailed specifications.

a) **Name all potential input and output parameters that can be used in the (vertex and fragment) shader by default (without being explicitly defined)?**

	Inputs	**Outputs**

Vertex Shader	`gl_BaseInstance`
	`gl_BaseVertex`
	`gl_DrawID`
	`gl_InstanceID`
	`gl_VertexID`
		`gl_Position` (Optional)
		`gl_ClipDistance`
		`gl_CullDistance`
		`gl_PointSize`

Fragment Shader	`gl_FragCoord`
	`gl_FrontFacing`
	`gl_PrimitiveID`
	`gl_SampleID`
	`gl_NumSamples`
	`gl_SamplePosition`
	`gl_SampleMaskIn`
		`gl_FragDepth`
		`gl_SampleMask`

Table 1: In- and Outputs for Vertex and Fragment Shader

For more Information see page 404ff for vertex shader in- and outputs and page 491ff for fragment shader in- and outputs.

b) **What are the reasons for this definition by default?**
OpenGL is a cross-platform graphics API (Application Programming Interface) that provides a means for applications to render 2D and 3D graphics. The definition of OpenGL includes the following reasons:

- Cross-platform compatibility: OpenGL allows for graphics rendering on multiple operating systems, including Windows, Linux, and macOS.
- High-performance graphics: OpenGL is optimized for high-speed rendering and is commonly used in demanding applications such as video games, simulations, and scientific visualizations.
- Wide industry adoption: OpenGL is widely adopted across various industries, including entertainment, automotive, and aerospace, due to its high-performance capabilities and cross-platform compatibility.
- Large developer community: OpenGL has a large and active developer community, which contributes to its continuous improvement and evolution.
- Strong backwards compatibility: New versions of OpenGL are backwards-compatible, meaning that applications written for older versions of the API will still work on newer implementations.

GLSL (OpenGL Shading Language) is a high-level shading language used in OpenGL to define the appearance of 3D graphics. The following are some reasons for the definition of GLSL:

- Flexible and powerful shading: GLSL allows for the creation of complex, highly detailed shaders, enabling advanced graphics effects such as reflections, shadows, and lighting. -platform compatibility: GLSL is supported on multiple platforms, including Windows, Linux, and macOS, making it a cross-platform solution for shader development.
- Real-time rendering: GLSL is optimized for real-time rendering, allowing for fast and efficient shading of 3D graphics.
- Easy integration with OpenGL: GLSL is closely tied to OpenGL, making it easy for developers to integrate shaders into their OpenGL applications.
- Active developer community: GLSL has a large and active developer community, with many resources and tutorials available to help developers learn the language and create advanced graphics effects.

c) **Name at least 3 built-in parameters in GLSL, and argue for their existence.**

(a) The variable `gl_Position` is intended for writing the homogeneous vertex position. It can be written at any time during shader execution. This value will be used by primitive assembly, clipping, culling, and other fixed functionality operations, if present, that operate on primitives after vertex processing has occurred. Its value is

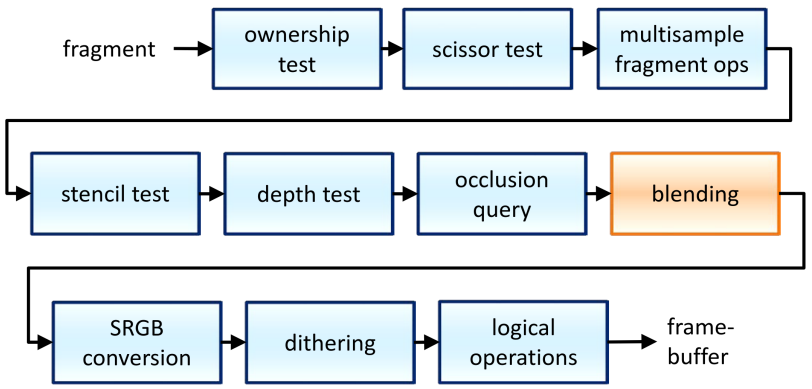

Figure 1: From Fragment Shader to Frame Buffer

undefined after the vertex processing stage if the vertex shader executable does not write `gl_Position`.

(b) The variable `gl_PointSize` is intended for a shader to write the size of the point to be rasterized. It is measured in pixels. If `gl_PointSize` is not written to, its value is undefined in subsequent pipe stages.

(c) The variable `gl_ClipDistance` is intended for writing clip distances, and provides the forward compatible mechanism for controlling user clipping. The element `gl_ClipDistance[i]` specifies a clip distance for each plane i. A distance of 0 means the vertex is on the plane, a positive distance means the vertex is inside the clip plane, and a negative distance means the point is outside the clip plane. The clip distances will be linearly interpolated across the primitive and the portion of the primitive with interpolated distances less than 0 will be clipped.

For more information see page 138ff

d) **Is it possible to overload functions or parameters in shaders?**

Yes, it is possible to overload functions or parameters in shaders.

2 Fragment Processing

In the lecture, you learned about multiple steps that are performed between the fragment shader and the frame buffer. For further reading, consult the OpenGL Wiki.

a) **Name each of these steps and shortly describe its functionality.**

- **Ownership test**: Because the Default Framebuffer is owned by a resource external to OpenGL, it is possible that particular pixels of the default framebuffer are not owned by OpenGL. And therefore, OpenGL cannot write to those pixels. Fragments aimed at such pixels are therefore discarded at this stage of the pipeline.
  Generally speaking, if the window you are rendering to is partially obscured by another window, the pixels covered by the other window are no longer owned by OpenGL and thus fail the ownership test. Any fragments that cover those pixels will be discarded. This also includes framebuffer clearing operations.
  Note that this test only affects rendering to the default framebuffer. When rendering to a Framebuffer Object, all fragments pass this test.
- **Scissor Test**: A rectangular area of the destination framebuffer can be designated as the only valid area for rendering to. All fragments aimed at pixels outside of this rectangle will be discarded at this stage.
- **Multisample Fragment Ops**: Multisampling is a process for reducing aliasing at the edges of rasterized primitives; Aliasing, antialiasing, coverage, primitive edges, fixed sample locations, multisample resolve, per-sample shading, smooth antialiasing
- **Stencil Test**: The stencil test, when enabled, can cause a fragment to be discarded based on a bitwise operation between the fragment's stencil value and the stencil value stored in the current Stencil Buffer at that fragment's sample position. This allows the user to lay down stencil values in one rendering pass, then conditionally cull fragments based on this pattern.
- **Depth Test**: The depth test, when enabled, allows a fragment to be culled based on a conditional test between the fragment's depth value and the depth value stored in the current Depth Buffer at that fragment's sample position. This is useful to cause geometry to be hidden behind other geometry. The closer geometry lays down depth values that mask the rendering of any fragments behind them, using the proper depth test condition.
- **Occlusion Query**: Occlusion queries are the collective name for the following query object types: `GL_SAMPLES_PASSED`, `GL_ANY_SAMPLES_PASSED`, and `GL_ANY_SAMPLES_PASSED_CONSERVATIVE`. These represent ways to detect whether an object is visible.
  Specifically, these queries detect whether any fragments continue being processed after reaching the depth test stage in the Per-Sample Processing part of the rendering pipeline. These operations proceed in the order defined on that page, so if a fragment passes the depth test, then it must also have passed all of the operations before it. So fragments which pass the depth test must also have passed the stencil test, scissor test, etc.
- **Blending**: Each of the colors in the fragment can be combined with the corresponding pixel color in the buffer that the fragment will be written to. The result of this blending operation is what will be written.

- **SRGB Conversion**: If `GL_FRAMEBUFFER_SRGB` is currently enabled, then for each fragment color that is being written to an image with an sRGB colorspace image format, the RGB components of the color are converted from linear to sRGB.
- **Dithering**: Dithering algorithms essentially fake a smooth gradient by varying which color is selected based on the position of the fragment. Therefore, if the color value is half-way between the two representable colors, then half of the pixels will be one color and half the other.
- **Logical Operations**: Fragment colors can be conditionally combined with the corresponding value in the framebuffer by performing boolean operations on them. This overrides Blending, and it only works for colors that are writing to integer (normalized or not) Image Formats that are not using sRGB writing to sRGB images.

b) **There have also been additional steps in older OpenGL versions. Name two of such old steps, explain their functionality, and describe how to achieve this functionality in modern OpenGL versions.**

- **Alpha Test:** Rejected fragments based on their alpha values. This can be achieved nowadays by implementing a conditional discard within the fragment shader. This has the advantage that arbitrarily complex conditions can be used and not only an alpha test.
- **Alpha Blending:** Combination of two images by means of an additional alpha image.

c) **Explain what depth peeling is and how it can be realized within the OpenGL pipeline.**

Depth peeling is a technique used in computer graphics to handle transparency in 3D scenes. The traditional depth sorting approach for handling transparency can produce visual artifacts, such as object order inconsistencies or the so-called "back-to-front" problem. Depth peeling solves these issues by rendering multiple depth layers of the transparent objects in the scene and combining them into a single image.
Depth peeling can be realized within the OpenGL pipeline the following way:

1) Initialize a list of transparent objects in the scene.
2) For each layer of transparency:
   a) Clear the color and depth buffers.
   b) Render the objects in the list that belong to the current layer, writing only the depth values to the depth buffer.
   c) Sort the objects in the list based on their depth value relative to the camera.
   d) Render the objects in the list again, but this time writing both color and depth values to the corresponding buffers.
   e) Repeat steps a to d until all transparent objects have been processed.

3) Blend the color and depth values from all layers to produce the final image.

This approach can produce more accurate results compared to traditional depth sorting, as it handles overlapping transparent objects correctly and reduces the visual artifacts caused by object order inconsistencies. However, it also comes with increased computational cost, as it requires multiple render passes to produce the final image.