Realtime–Rendering with OpenGL

The Graphics Pipeline

Francesco Andreussi

Bauhaus–Universität Weimar

14 November 2019

Bauhaus-Universität Weimar

Faculty of Media

Basic Information

- Class every second week, 11:00–12:30 in this room, every change will be communicate to you as soon as possible via mail
- Every time will be presented an assignment regarding the arguments covered by Prof. Wüthrich and me to fulfil for the following class. It is possible (and also suggested) to work in couples
- The submissions are to be done through push requests on Github and the work has to be presented in person scheduling an appointment with me

Delays in the submissions and cheats (e.g. code copied from the Internet or other groups) WILL NOT be accepted!

If you find code online **DO NOT COPY IT** but adapt and comment it, otherwise I have to consider it cheating and you will be rewarded with a 5.0

For any question you can write me at *francesco.andreussi@uni–weimar.de*

Graphics Processors & OpenGL

A 3D scene is made of primitive shapes (i.e. points, lines and simple polygons), that are processed by the GPU *asynchronously* w.r.t. the CPU. OpenGL is a **rendering library** that exposes some functionality to the "Application level" and translates the commands for the GPU driver, which "speaks" forwards our directives to the graphic chip.

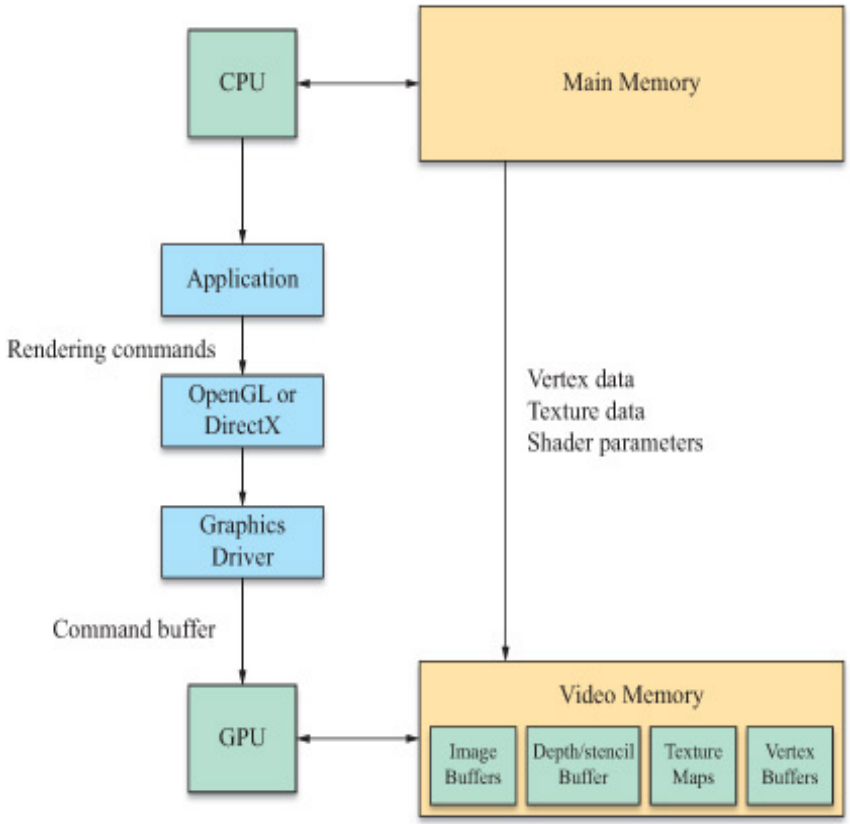

Fig. 1: Communications between CPU and GPU

OpenGL

OpenGL is an *open standard* and *cross-platform* API.

Its compatibility makes it widely used but can cause behavioural differences between platforms, in particular in case of mistakes or misuse.

This course will cover and use modern OpenGL (4.0, but every version $\geq$3.0 should work roughly the same). Older versions support the "immediate mode" which forces to use a lot of low-level functions, while now for these purposes we use *shaders*, helping to move many functionalities (and data) directly in the graphics card.

Actually, OpenGL has been superseded by Vulkan. Announced in 2015, only recently it has reached a significant market-share.

The Rendering Pipeline

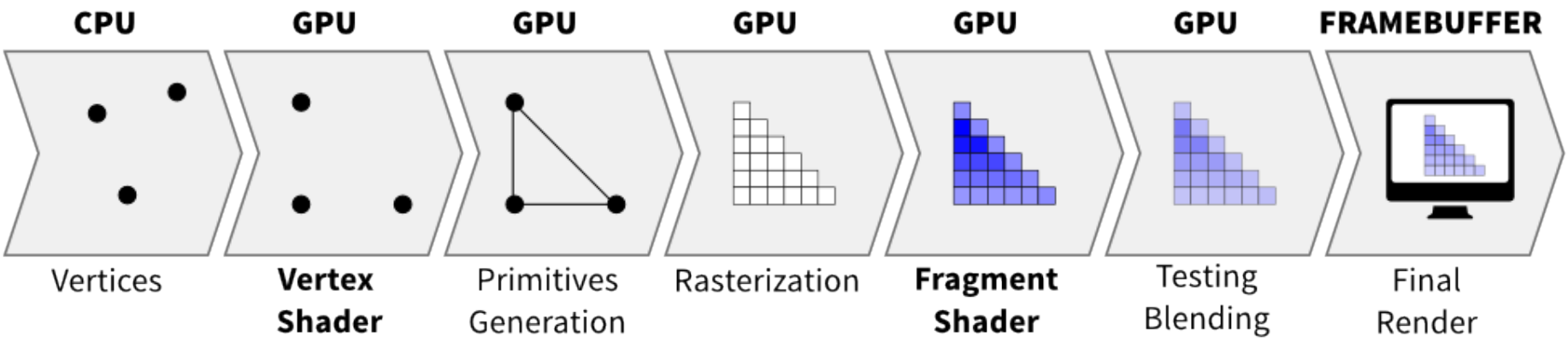

Fig. 2: The core stages of a Rendering Pipeline

In order to render 2D or 3D computer graphics, our data have to be processed going from vertices and attributes to coloured pixels in a viewport.

Geometry Stage

The Scene Graph

In order to be intuitive for the programmers **all** the **CG** related **applications** will probably implement and provide a **Scene Graph**, i.e. an **hierarchical** organisation of the objects in the scene.

Every node inherit the **coordinate system** of their ancestor. Thus, modifying the `localMatrix` attribute of a node, we can affect the position and location of all its descendants.

Then, every node defines its own **local** coordinate system even, and most of all, the root of this tree which defines the **world coordinate system**.

Geometry Stage

Vertex Processing & Clipping

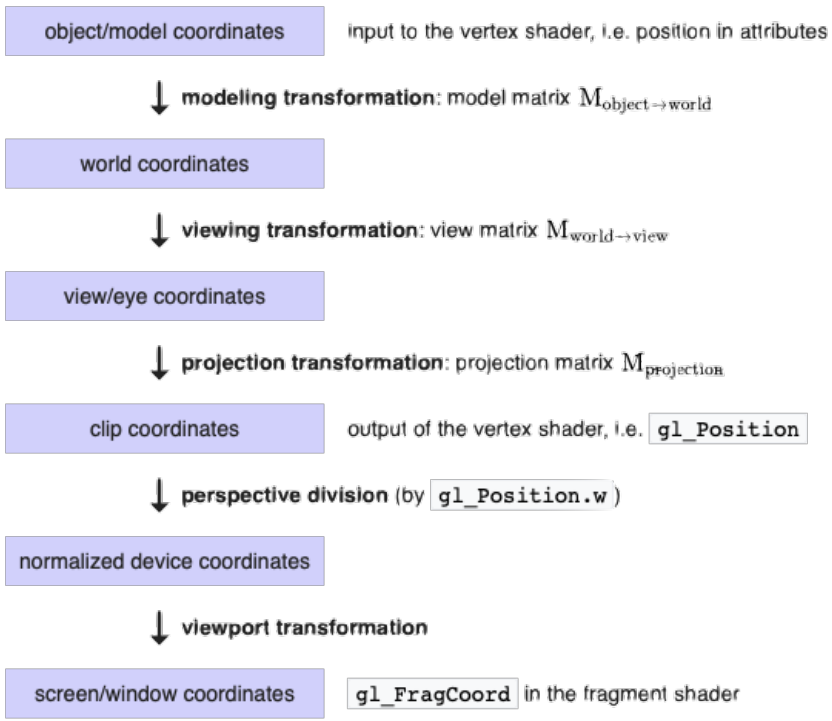

Fig. 3: The coordinate systems transformations

The **vertex processing** is handled by the programmable **vertex shader**. Its main task is to transform the local coordinate systems of every abject/model in the scene into the **clip** coordinate system.
The vertices in Clip Space are passed to the Vertex **Post-Processing** stage, were **clipping** (removing all the vertices that are not in the **frustum**), **perspective division** and **viewport transformation** are performed. Only now, **Primitives Generation** is executed.

Geometry Stage

The Frustum

The frustum defines the portion of the space that is framed by the camera. In order to have **perspective** we have to multiply the vertices in Eye/View/Camera Space by `gluPerspective` (easier to manage) or `glFrustum` (more flexible).

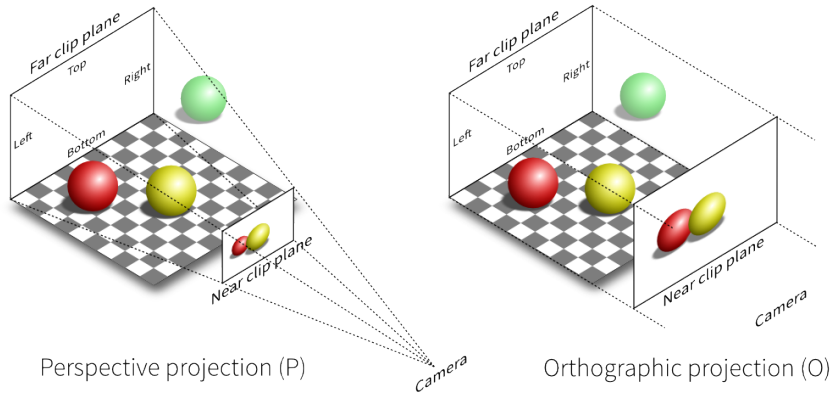

Fig. 4: A perspective and an orthogonal frusta

Otherwise, to have an **orthographic** view, we have to use the matrix `glOrtho`.
A frustum can be defined in two ways: `glOrtho` and `glFrustum` take near, far, left, right, top and bottom coords $(-n, -f, l, r, t, b)$ as params, while `gluPerspective` has only near, far planes, aspect ratio and vertical FOV $(-n, -f, \alpha, \theta_{fov\_y})$.

Geometry Stage

Tessellation & Geometry Shader

These two *programmable* shaders are *optional* and they are executed between the Vertex Shader and the Vertex Post-Processing.

Tessellation is subdivided in three steps:

- *TCS* (control): ensures continuity and coherence between patches — programmable
- *Tess. Primitive Generation*: it generates new vertices w.r.t. the input received from TCS — fixed
- *TES* (evaluation): compulsory for the execution of tessellation, similar to a vertex shader but it can generate new vertices too — programmable

Geometry Shader makes possible to modify dynamically at runtime the models, useful for *Displacement Mapping* and *Geometry Compression* ($\rightarrow$ Levels Of Detail).

Geometry Stage

Face Culling

Face Culling can be enabled, then the back–face of a triangle is discarded. OpenGL distinguishes front and back faces w.r.t. the ordering of the vertices when we define a triangle: **clockwise** $\rightarrow$ **front**, **counterclockwise** $\rightarrow$ **back**.

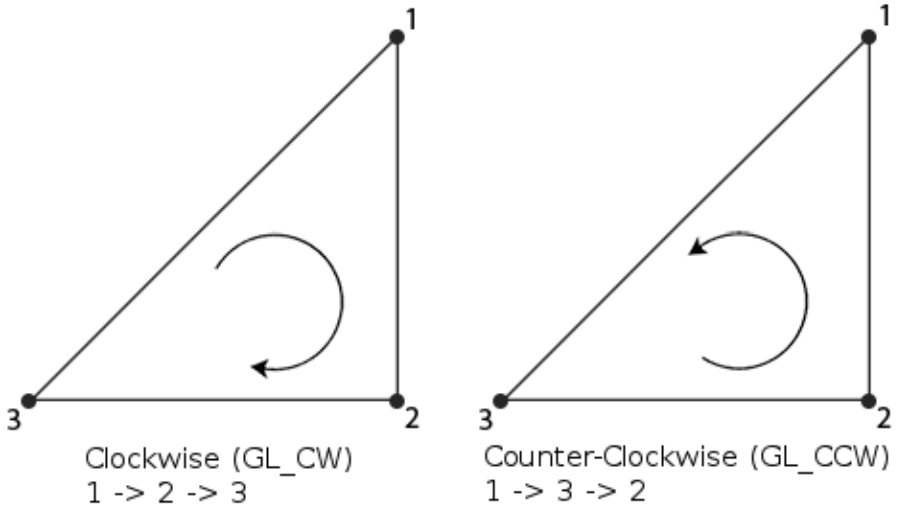

Fig. 5: Triangle order definition

Rasterising Stage

Rasterisation

It is the **core** of the pipeline, fixed and essentially unmodified through the years.

Now, the vertices are in Viewport Space: $x, y, z \in \mathbb{R}$, $0 \leq x \leq w_{screen}$, $0 \leq y \leq h_{screen}$, $-1 \leq z \leq 1$. However, displays have a finite number of pixels, then the GPU has to **discretize**.

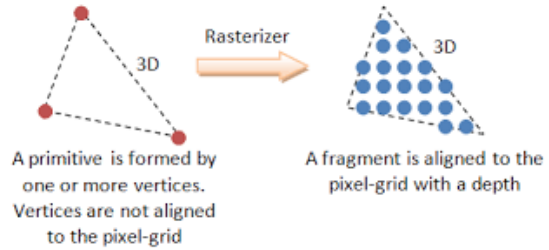

Fig. 6: Before/After Rasterisation

Drawing **vertices** is trivial (the corresponding pixel is painted), for **lines** there are a number of techniques, all based on scanning horizontally or vertically the pixels between the vertices and assign the interpolated attributes to the "best approximation" of the line, finally, for **triangles**, simplifying a lot, the discretised edges are calculated and then all the pixels between them receive the *bilinearly* interpolated attributes of the vertices.

Rasterizing Stage

Fragment Shader

Arguably, the main part of the Computer Graphics program. It is executed for each fragment in the scene and takes as (main) inputs:

- `gl_FragCoord`: `vec4` storing the fragment location in Window Space,
- `gl_FrontFacing`: `bool` saying if it belongs the front face of a triangle (for lines and points always `true`).

It outputs the fragment colour and, optionally, `gl_FragDepth` (by default is equal to `gl_FragCoord.z`) and the `gl_SampleMask` (useful for Multi–sampled Rendering). Thanks to the Fragment Shader efficient and high–quality lightning and texturing are possible.

P.S: the fragment to pixel ratio is ≥ 1. Objects can overlap and the rasterisation processes also the hidden parts of a scene.

Rasterizing Stage

Buffer Tests & Post-Processing (1)

- **Pixel Ownership**: checks whether the OpenGL application is in the foreground for a certain pixel or not. If not the all the fragments of that pixel are discarded,

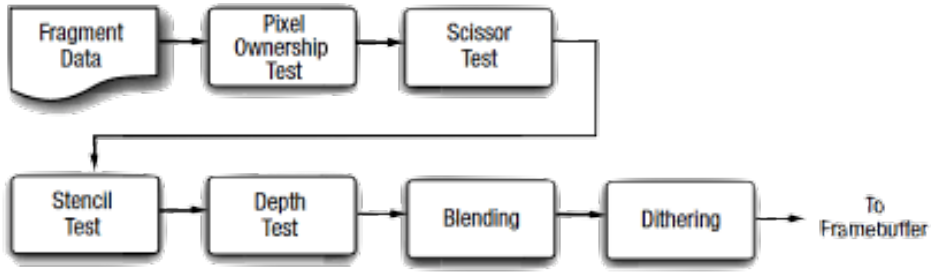

Fig. 7: Per–Fragment operations Pipeline

- **Scissor Test**: throws away the fragments outside a user–defined area of the viewport,
- **Alpha Test**: compares the alpha value of a fragment (i.e. its transparency) to a user-defined constant, if test fails the fragment will not be rendered,
- **Stencil Test**: test between an attribute of a fragment and a value stored in the *stencil buffer*,

Rasterizing Stage

Buffer Tests & Post-Processing (2)

- **Depth Test**: discards a fragment if a test between the final fragment depth and the corresponding value stored in a *depth buffer* fails,
- **Blending**: the final pixel colour is computed combining the colours of all the fragments in that position, taking in account vorious properties, such as alpha,
- **Ditching**: it, essentially, converts the computed colours from a high–precision to a lower–precision space.

The **Post-Processing** it is useful if we have to apply an effect to the whole scene (B/W, sepia, blur fx...). The basic idea is to take the **Frame Buffer** and use it as a texture: modify it with a shader and then map it on the window.

The Application

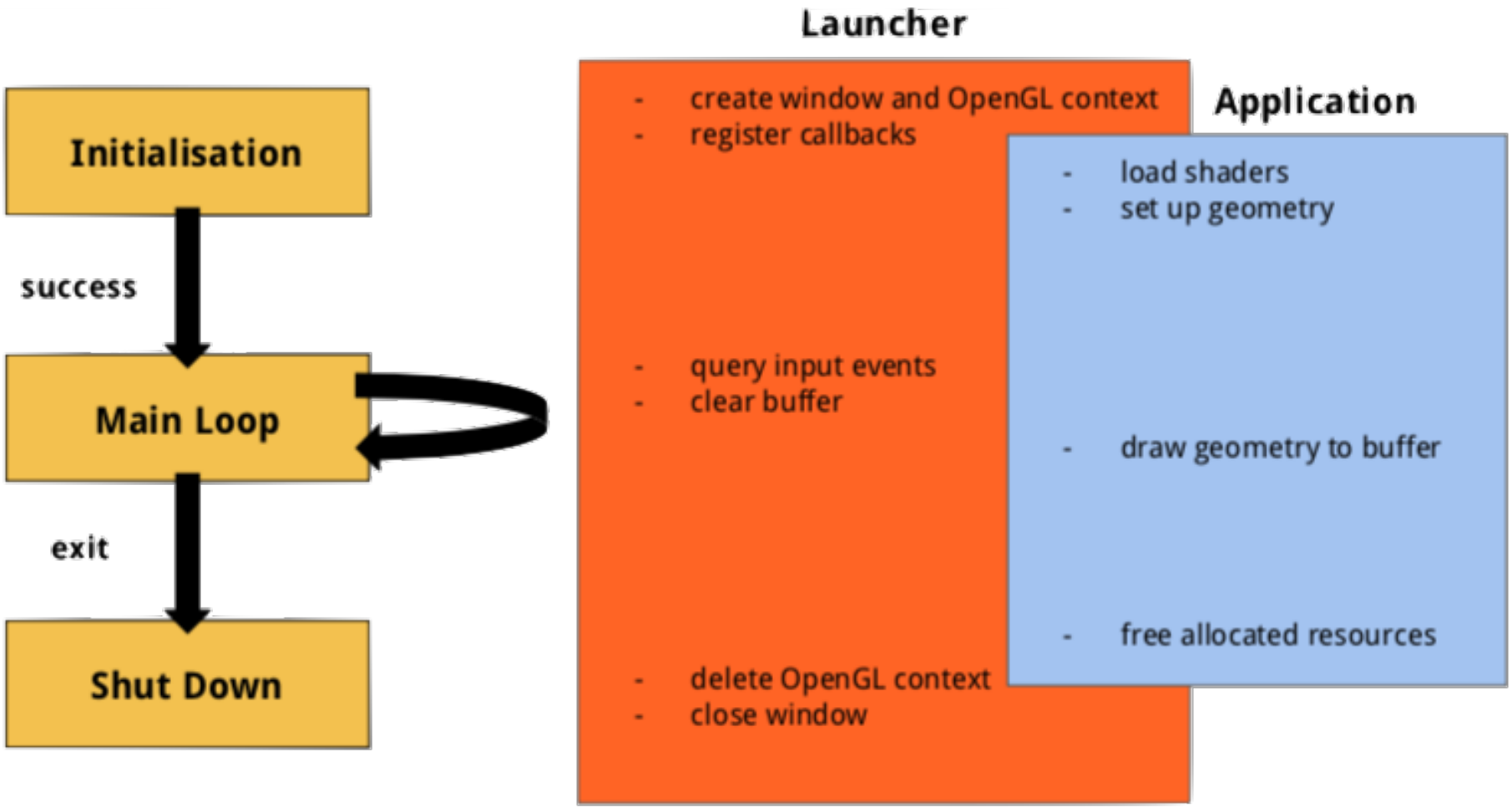

Fig. 8: Graphics application workflow

Useful Links

Offical online resources:

- *OpenGL Wiki*
- *API Reference Guide PDF (v4.6)*
- *OpenGL Reference Pages*
- *glBinding documentation*
- *glfw documentation*
- *glm documentation*

Further online OpenGL tutorials/books:

- *Wikibooks*
- *opengl-tutorial.org*
- *learnopengl.com*
- *open.gl*