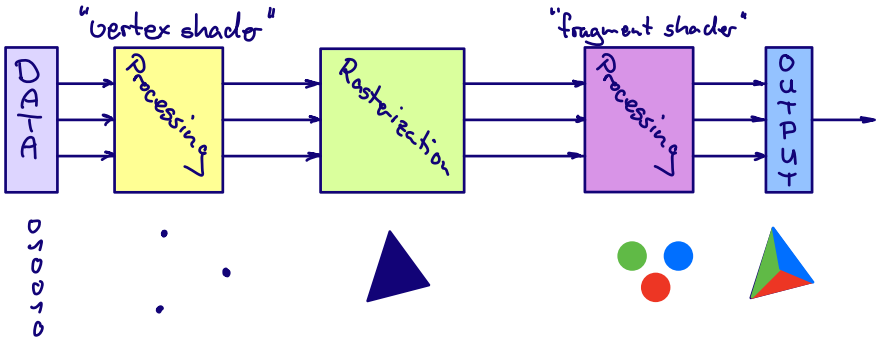
OpenGL

is an (not object oriented) API for GPU programming

-> massively parallel processing (each shader is a specific pice of hardware)

Basic Pipeline

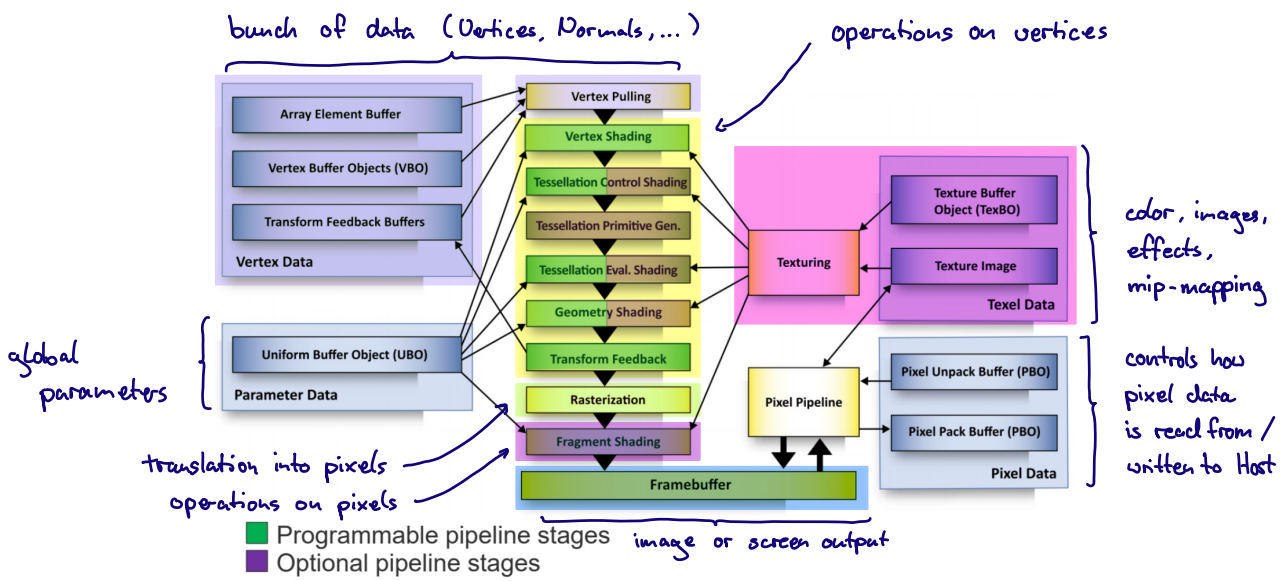


-> processing steps are programmable and provide much more opportunities

Creating Shader Programs

- code each (needed) stage (shader) seperately
- create a shader program, call all shaders and link them

OpenGL Overview



Vertex Pulling (Vertex Data)

incoming data is unorganised.

- **Vertex Buffer Object** stores data for specific purpose, e.g. points, color, surface normals
  → Vertex Buffer Objects are united as **Vertex Array Object** for processing, which forwards the data to the shader

Vertex Shader

- executed once per vertex
- only can use **vertex data** and **uniform (global) parameters**
- processes only a particular point, e.g. assigning color, or texture coordinates to it

Texturing

- Load texture image
- set up **texture sampler configuration** (magnification, minification, mip-mapping)
- bind configuration to **texture unit**
- returns texture coordinates (to vertex data) and texture unit number as uniform variable
- rasteriser interpolates over texture color

Tesselation

render curved or displaced surfaces

- **Tesselation Control Shader** specifies in how many subdivisions a triangle should be separated (separation points per edge)
  Tesselation Level:
  - outer = how often subdivide edge
  - inner = how many primitives within a triangle
- **Tesselation Primitive Generation** produces the triangles
- **Tesselation Evaluation Shader** creates correct positions and colors/texture points

Geometry Shader

- executed once per primitive (point, line, triangle)
- can use the whole vertex data of the primitive
- returns flexible amount of primitives (vertices and color), eg. for a given point it returns 4 points representing a rectangle at the given point

Transform Feedback

- in latest stage before rasterisation
- data written to transform feedback buffers → used as input for vertex shader
  → buffer has dynamic length
- Transform Feedback Objects group all transform feedback buffers written by one shader

Fragment Shader

- executed once per rendered pixel
- interpolates vertex data from previous programmable stages

Framebuffer

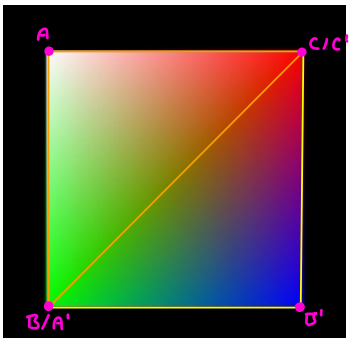
- Rendertarget
- it's possible to assign a render target to a texture
  ↳ can create pictures in pictures, using the Pixel Pipeline

Benefits of OpenGl compared to Ray Tracing

→ it uses rasterisation which is much faster
→ it directly uses hardware shaders (also improve of speed)
→ parallel computation more effective than in Ray-Tracing

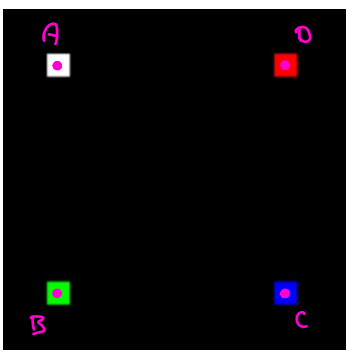
OpenGL Examples from the Lecture

Colored Quad



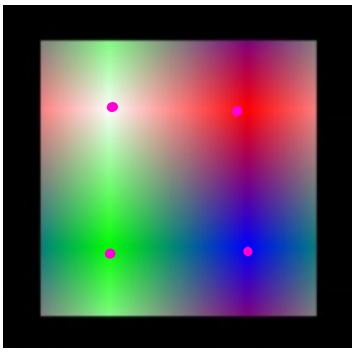
- Data: 6 Points, 4 Colors (2 Points are redundant)
- Stages:
  1. Vertex Pulling
  2. Vertex Shading
  3. Rasterisation
  4. Fragment Shading
  5. Frame buffer
- Process:
  1. Vertex Shader assigns color to vertices
  2. Rasterisation: triangles defined, "pixeling" the triangles
  3. Fragment Shader interpolates color for each pixel, based on position in triangle (Barycentric Coordinates)

Point Sprites



- Data: 4 Points, 4 Colors
- Stages:
  1. Vertex Pulling
  2. Vertex Shading
  3. Geometry Shading
  4. Rasterisation
  5. Fragment Shading
  6. Frame buffer
- Process:
  1. Vertex Shader assigns color to each point
  2. Geometry shader calculates squres for each point (vertices and color)
  3. Rasterisation
  4. Fragment Shader interpolates color for each square (two triangles)

Texture Quad (Texture is set of 4 colored points)

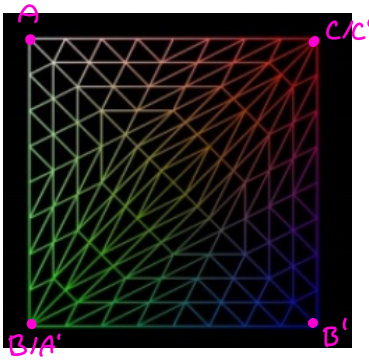


- Data: 6 Points, 6 Texture Coordinates, Texture Image
- Stages:
  1. Vertex Pulling
  2. Texture Sampling
  3. Vertex Shading
  4. Rasterisation
  5. Fragment Shading
  6. Frame buffer
- Process:
  1. create texture and sample configuration
  2. Vertex Shader assigns texture coordinates to points
  3. Rasterisation
  4. Fragment Shader interpolates between points → continuous representation (assumes repeating pattern of texture)

Animated Sprites

- Data: 4 Points, 4 Colors, 4 Velocity Vectors
- Stages:
  1. Vertex Pulling
  2. Vertex Shading
  3. Geometry Shading
  4. Transform Feedback
  5. (~~Rasterisation~~)
  6. (~~Fragment Shading~~)
  7. Frame buffer
- Process (recursive):
  1. Vertex Shader (gets data from UBO or Transform Feedback Buffer) → computes new position of points
  2. Geometry Shader → squares
  3. Transform Feedback stores data to Buffer
  4. Rasterization and Fragment Shader disabled → the hell, why ?!?

Tesselated Quad



- Data: 6 Points, 4 Colors
- Stages:
  1. Vertex Pulling
  2. Vertex Shading
  3. Tessellation Control Shading
  4. Tessellation Primitive Gen.
  5. Tessellation Eval. Shading
  6. Rasterisation
  7. Fragment Shading
  8. Frame buffer