Rendering

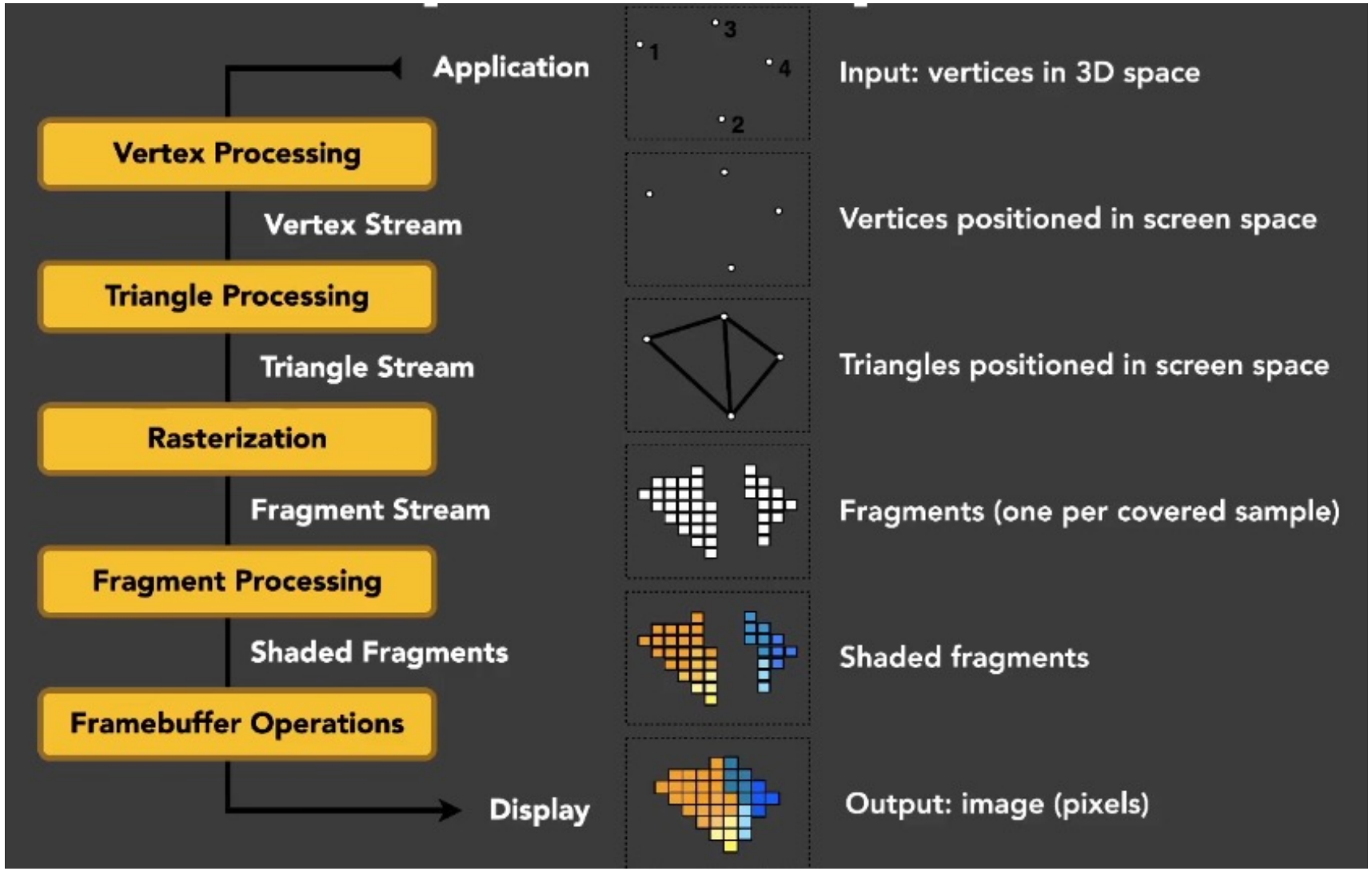
Sources

- GAMES202 Real-Time High Quality Rendering
- Real-Time Rendering, 4th edition

Introduction

- Real-Time: 30+ FPS
  Interactivity: frame generated on the fly
  High Quality: realism, dependavlity(correctness)
  Rendering: 3D scene -(calculating light)> image
- 4 parts:
  - shadows (and env)
  - global illumination(scene/image space, precomputed)
  - physically-based shading
  - real-time tracing
  - ...

Graphics (Hardware) Pipeline



OpenGL

- Summary: in each pass
  - Specify objects, camera, MVP, etc.
  - Specify framebuffer and input/output textures
  - Specify vertex / fragment shaders
  - (When you have everything specified on the GPU) Render!

- A. Place objects/models
  - Model specification
  - Model transformation

- User specifies an object's vertices, normals, texture coords and send them to GPU as a Vertex buffer object (VBO)
  - Very similar to .obj files

- Use OpenGL functions to obtain matrices
  - e.g., glTranslate, glMultMatrix, etc.
  - No need to write anything on your own

OpenGL

- B. Set up an easel
  - View transformation
  - Create / use a framebuffer
- Set camera (the viewing transformation matrix) by simply calling, e.g., gluPerspective

```
void gluPerspective( GLdouble fovy,
                     GLdouble aspect,
                     GLdouble zNear,
                     GLdouble zFar );
```

OpenGL

- C. Attach a canvas to the easel
- Analogy of oil painting:
  - E. you can also paint multiple pictures using the same easel
- One rendering pass in OpenGL
  - A framebuffer is specified to use
  - Specify one or more textures as output (shading, depth, etc.)
  - Render (fragment shader specifies the content on each texture)

One frame buffer
Multiple outputs.

MRT->Multi-Render Target

OpenGL

- D. Paint to the canvas
  - i.e., how to perform shading
  - This is when vertex / fragment shaders will be used
- For each vertex in parallel
  - OpenGL calls user-specified vertex shader:
    Transform vertex (ModelView, Projection), other ops
- For each primitive, OpenGL rasterizes
  - Generates a *fragment* for each pixel the fragment covers

OpenGL

- For each fragment in parallel
  - OpenGL calls user-specified fragment shader: Shading and lighting calculations
  - OpenGL handles z-buffer depth test unless overwritten
- This is the “Real” action that we care about the most: user-defined vertex, fragment shaders
  - Other operations are mostly encapsulated
  - Even in the form of GUI-s

Shadow-light

- Now what’s left?
  - F. Multiple passes! (Use your own previous paintings for reference)

OpenGL Shading Language (GLSL)

- Vertex/fragment shading described by small program
- Written in language similar to C but with restrictions

Shader Setup

- Initializing (shader itself discussed later)
  - Create shader (Vertex and Fragment)
  - Compile shader
  - Attach shader to program
  - Link program
  - Use program

Shader source is just sequence of strings

Similar steps to compile a normal language