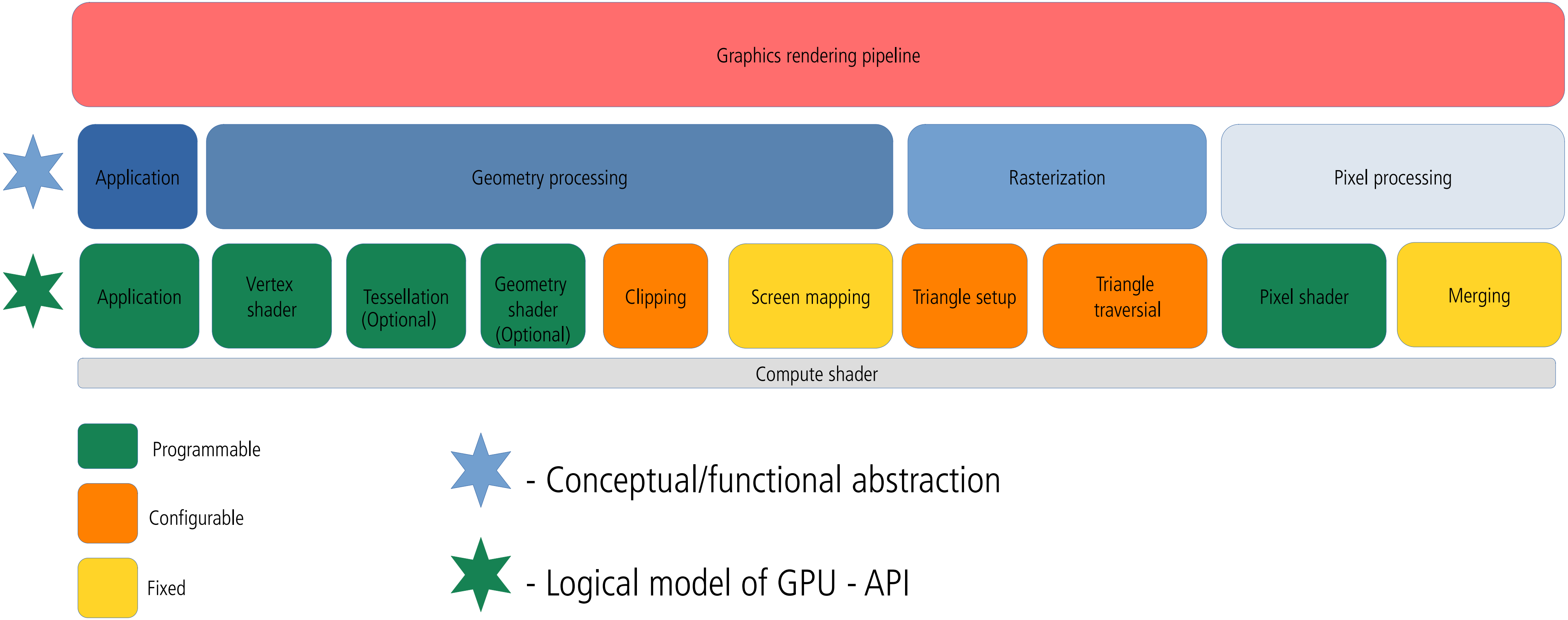
Graphics rendering pipeline – logical model of GPU

Pipeline abstraction

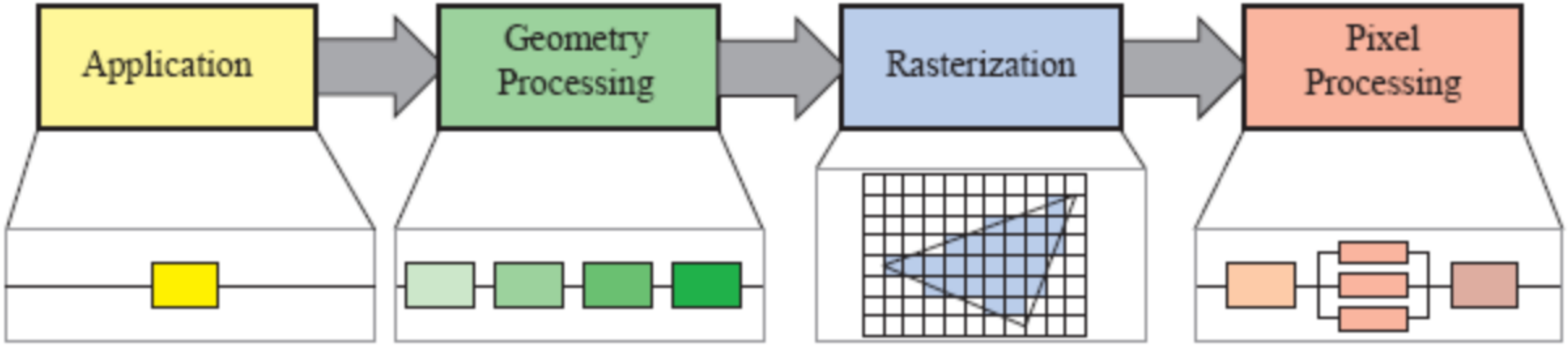
- **Functional/conceptual stages –** task to be performed but not how
  - **Physical/Implemented stages –** how are functional stages implemented in hardware and exposed to the user as API.
- **Logical model of GPU** – exposed to a programmer by API. Physical model is up to the hardware vendor.

Graphics rendering pipeline overview



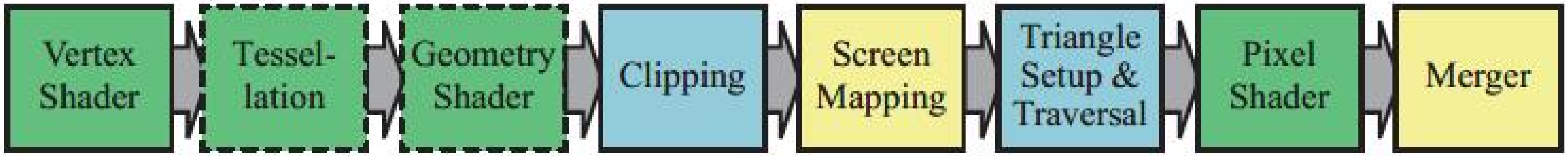
GPU graphics rendering pipeline

- GPU implements conceptual rendering pipeline



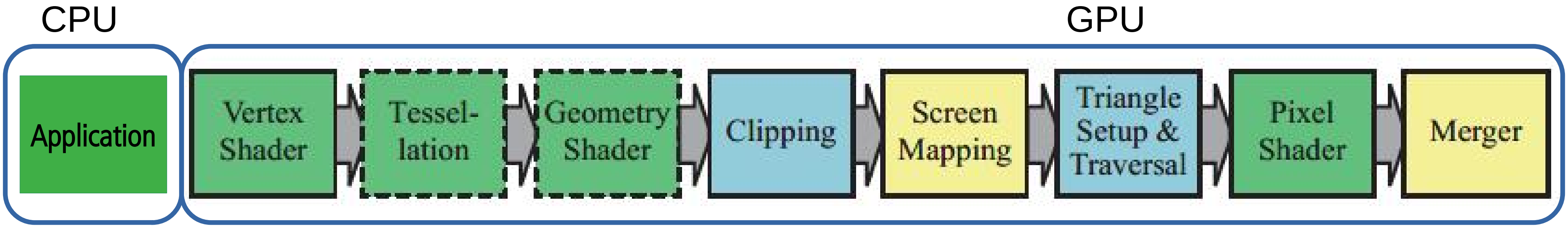
GPU graphics rendering pipeline

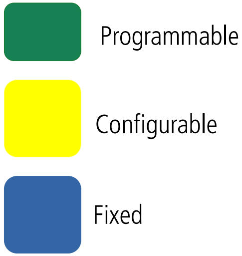
- Logical model is exposed to used as API
  - Underlaying hardware implementation varies by hardware vendor (e.g., Nvidia, AMD, Intel, etc.)



Programming a GPU renderer

- As programmers, we are interested in:
  - Application stage
  - GPU programmable and configurable stages: vertex shader, pixel shader, screen mapping and merger





Programming a GPU renderer

- **Unified shader design**: vertex, pixel, geometry, tessellation shaders share common programming model – same instruction set arhitecture (ISA)
- GPU with cores which supports unified shader design: **unified shader architecture**

Programming a GPU renderer

- Application stage can be written in various languages:
  - C, C++
  - Python
  - Rust
  - etc.
- Different APIs enable GPU rendering:
  - OpenGL (cross-platform), OpenGL shading language (GLSL)
  - DirectX (Windows), High-level shading language (HLSL)
  - Vulkan (cross-platform)
  - Metal (iOS)

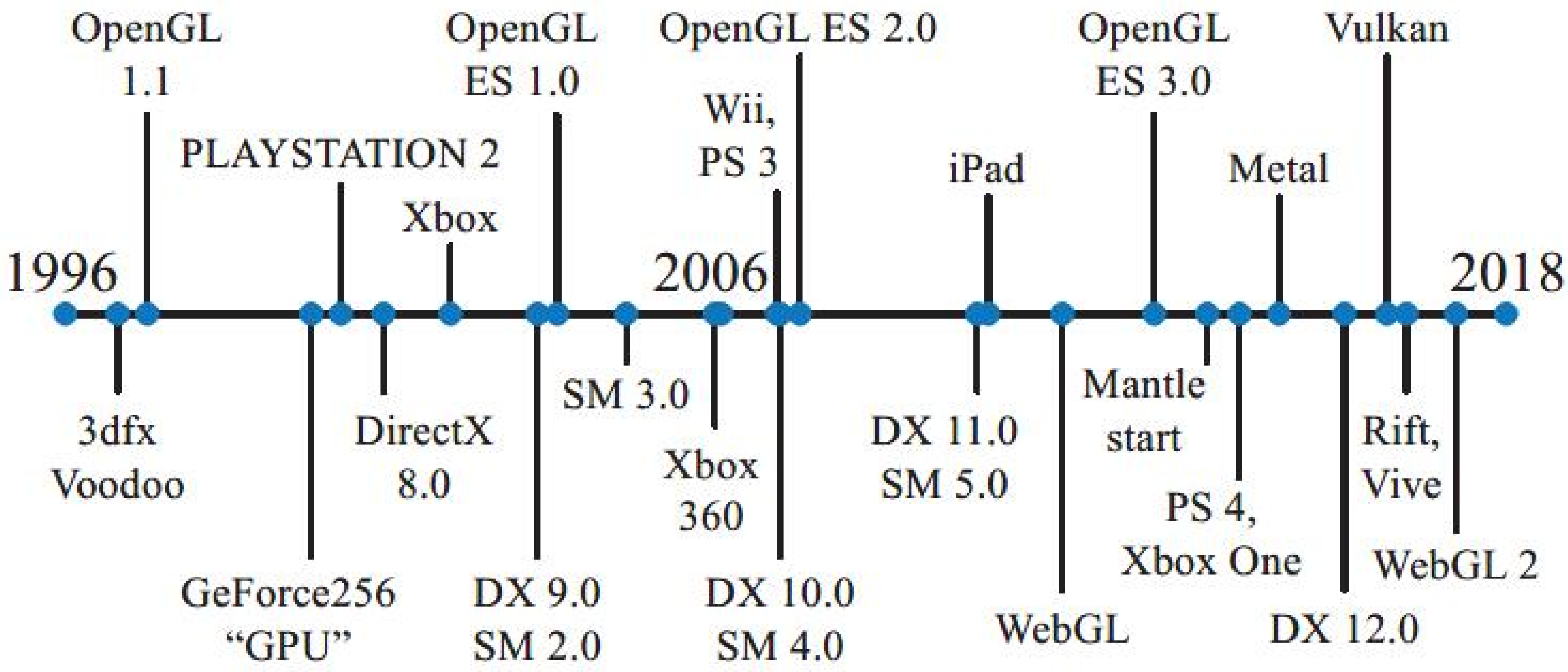
Programming a GPU renderer

- Parameters and states of graphics API are defined on application stage
- **Draw call** invokes graphics API to draw primitives
  - Causing graphics pipeline to execute and run its shaders
- Programmable shaders have two types of inputs:
  - **Uniform inputs**: values remain the same through draw call, e.g., transformation matrix of static object, color of light source, texture, etc.
  - **Varying inputs**: data that come from triangle vertices or rasterization, e.g., triangle surface location changes per pixel

Shading languages

- Vertex, geometry, tessellation shader
  - Enables programming of geometry behaviour
- Fragment shader
  - Enables programming material models, light interaction, post-processing effects
- Shading languages support common graphics computations:
  - Additions, multiplications, etc.
  - Intrinsic functions: cos(), atan(), log(), etc.
  - Complex operations: vector normalization, reflection, cross product, matrix operations, etc.
  - Flow control on uniforms (static flow control) and varyings (dynamic control flow)

Evolution of graphics API and hardware

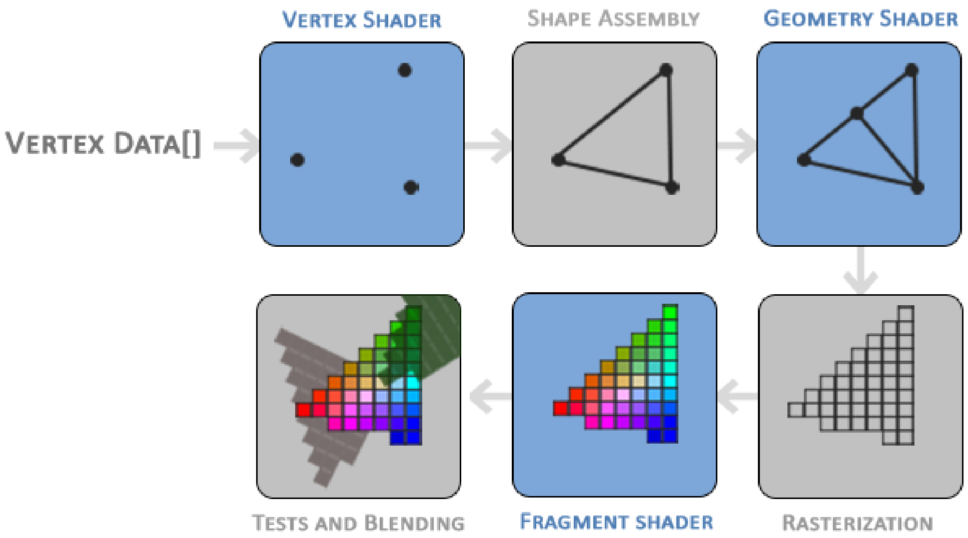


https://www.realtimerendering.com/

Graphics rendering pipeline – OpenGL demo

OpenGL rendering pipeline

- Application stage
- Vertex shader
- Fragment shader



Application stage

- Define:
  - Camera
  - Light
  - Object: geometry and material
  - Shader programs
  - Configurable GPU pipeline parameters
- Draw call

https://learnopengl.com/code_viewer_gh.php?code=src/2.lighting/2.2.basic_lighting_specular/basic_lighting_specular.cpp

Application stage: camera

- Resolution
- Position, orientation
  - View and projection matrix

```
// settings
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;
```

```
// camera
Camera camera(glm::vec3(0.0f, 0.0f, 3.0f));
```

```
// view/projection transformations
glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
glm::mat4 view = camera.GetViewMatrix();
lightingShader.setMat4("projection", projection);
lightingShader.setMat4("view", view);
```

Application stage: light

- Point light:
  - Position
  - Color

```
// lighting
glm::vec3 lightPos(1.2f, 1.0f, 2.0f);
```

Application stage: object

Vertex shader

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;

out vec3 FragPos;
out vec3 Normal;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = mat3(transpose(inverse(model))) * aNormal;

    gl_Position = projection * view * vec4(FragPos, 1.0);
}
```

https://learnopengl.com/code_viewer_gh.php?code=src/2.lighting/2.2.basic_lighting_specular/2.2.basic_lighting.vs

Fragment shader

https://learnopengl.com/code_viewer_gh.php?code=src/2.lighting/2.2.basic_lighting_specular/2.2.basic_lighting.fs

```
#version 330 core
out vec4 FragColor;

in vec3 Normal;
in vec3 FragPos;

uniform vec3 lightPos;
uniform vec3 viewPos;
uniform vec3 lightColor;
uniform vec3 objectColor;

void main()
{
    // ambient
    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * lightColor;

    // diffuse
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;

    // specular
    float specularStrength = 0.5;
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
    vec3 specular = specularStrength * spec * lightColor;

    vec3 result = (ambient + diffuse + specular) * objectColor;
    FragColor = vec4(result, 1.0);
}
```

Output