GDC | March 20-24, 2023
San Francisco, CA

Render Graph

A Data Oriented Approach

by Zhenglong ZHOU & Huabin LING
Cocos Technologies

#GDC23

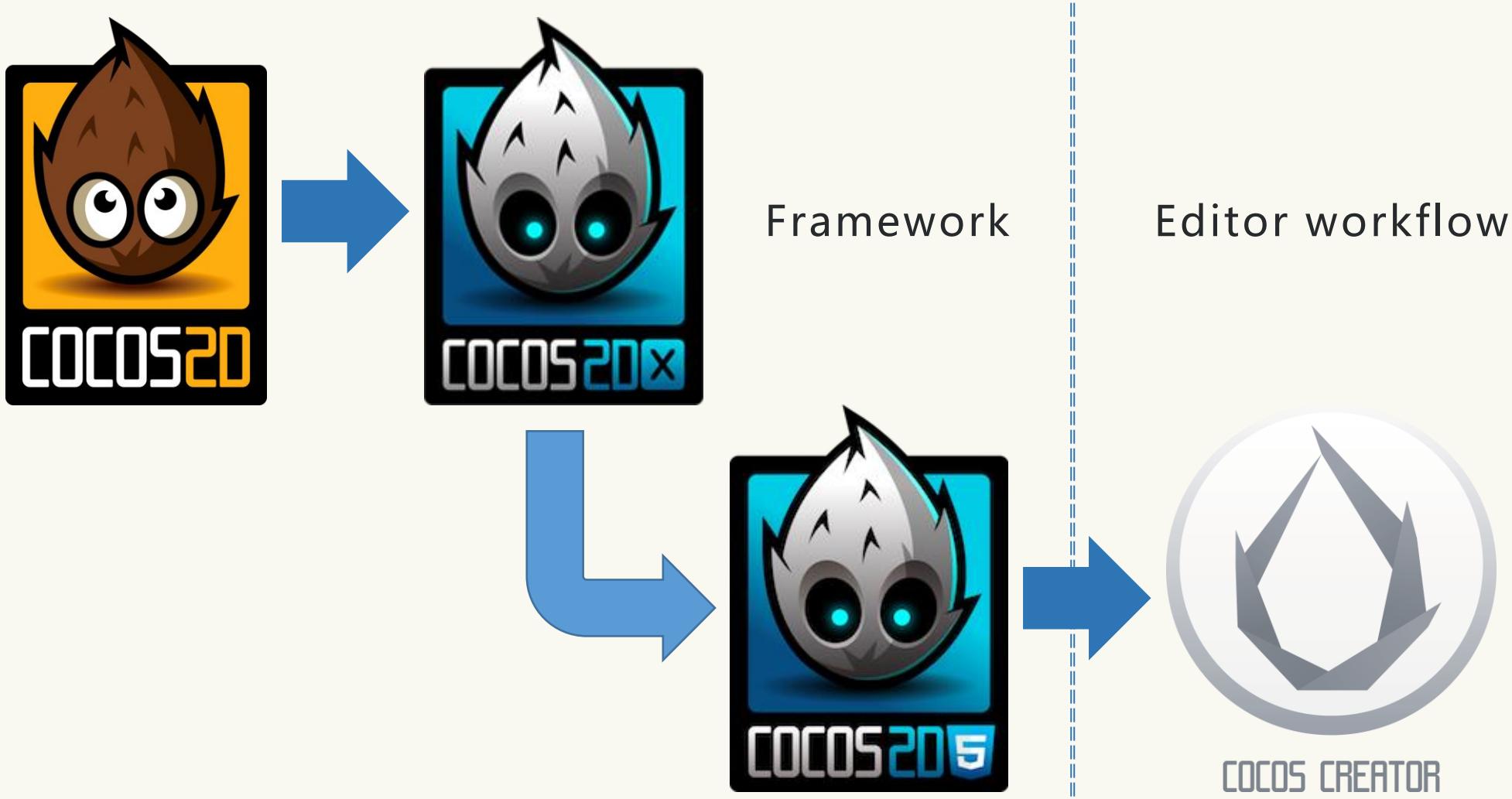
- Huabin LING (aka panda)
  - Technical Director of Cocos Engine
  - Building engine team of Cocos

- Dr. Zhenglong ZHOU
  - Render Pipeline Architect
  - Love game engine programming

CONTENTS

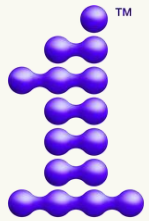
1. Cocos: Build an Open Source Engine
2. Open Source Sample Project
3. Design Context of Render Graph
4. Core Design Explained

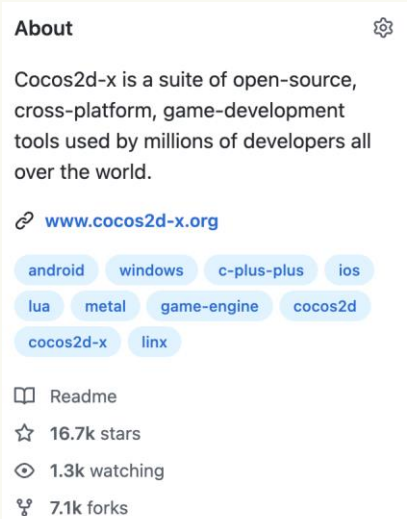
Cocos: A Cross Platform Open Source Engine



How Open Source Helped Us

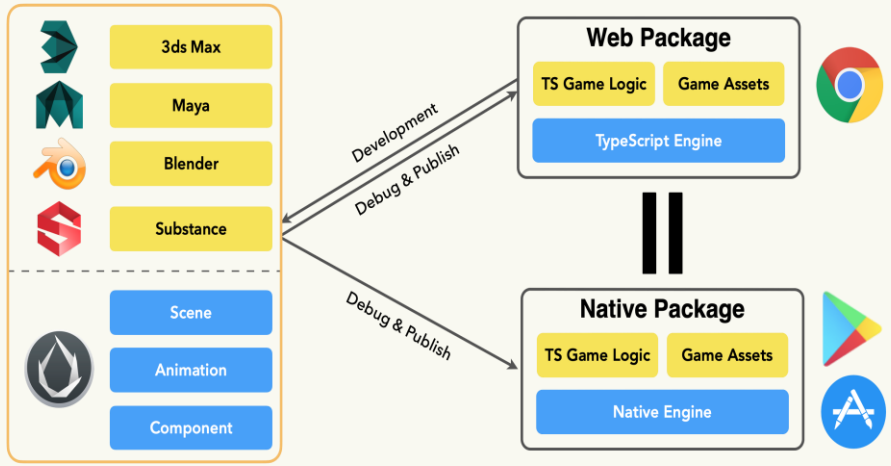
- Cocos2d-x
  - 16.7K stars / 7.1K forks / 624 contributors
- Cocos Creator engine
  - 3.4K stars / 1.3K forks / 135 contributors



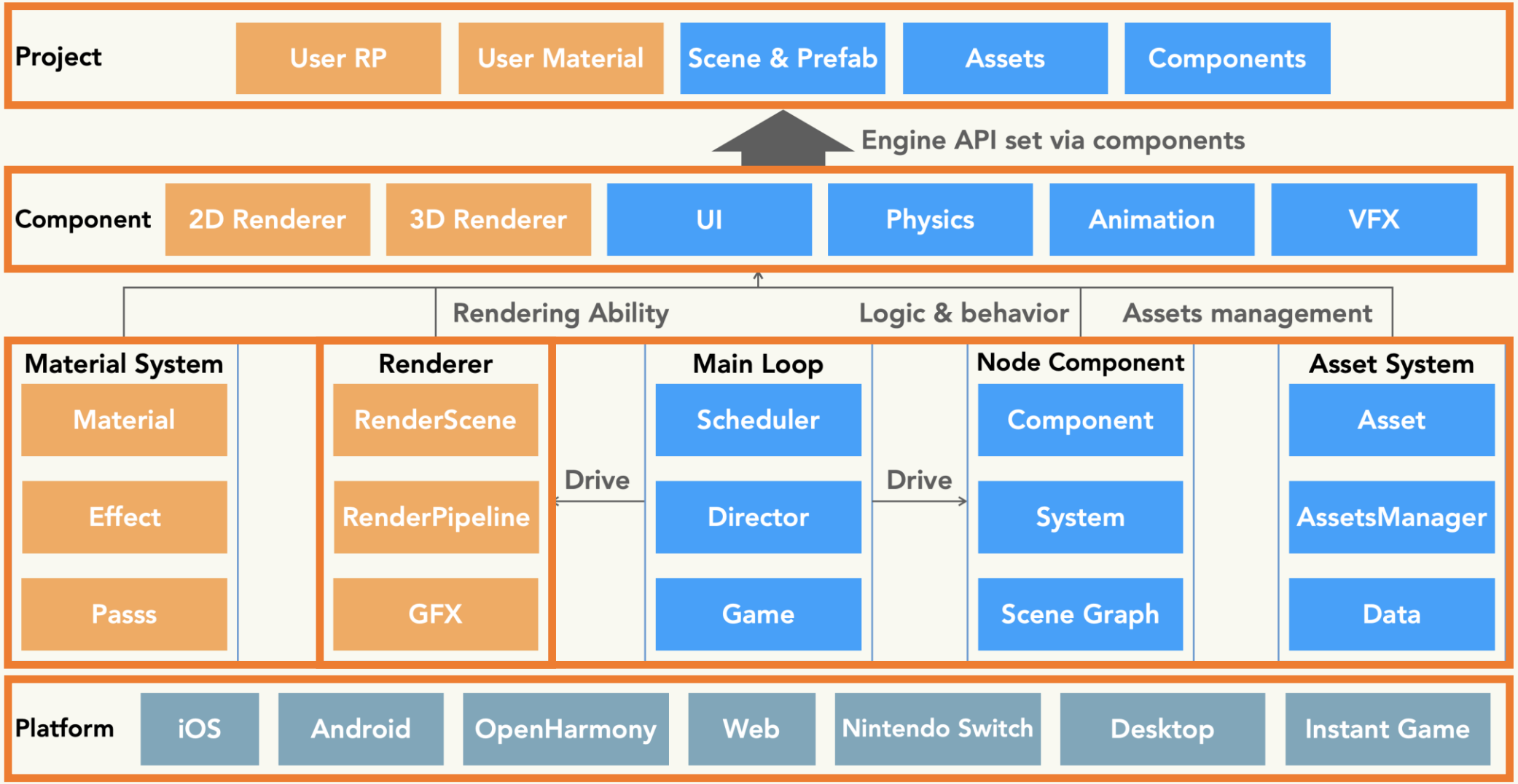


How Open Source Helped Our Developers

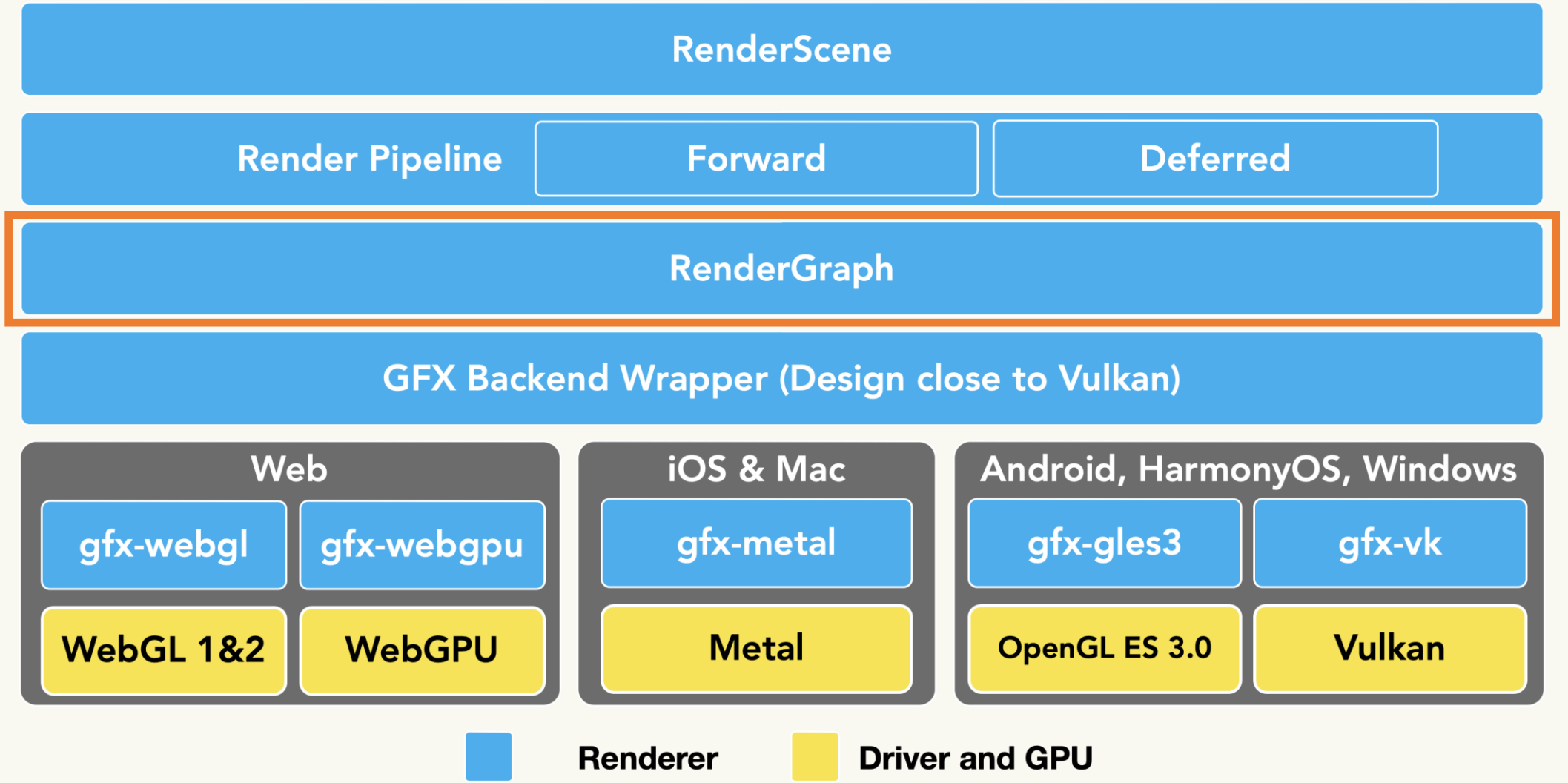
- Better understanding of the engine, for debugging or learning
- Easy extending the engine with needed features
- The engine isn’ t perfect, but users feel they are in control
- Knowing the roadmap and direction, participating in it
- Building trust with total transparency



Overall Engine Architecture



Focusing on the Renderer

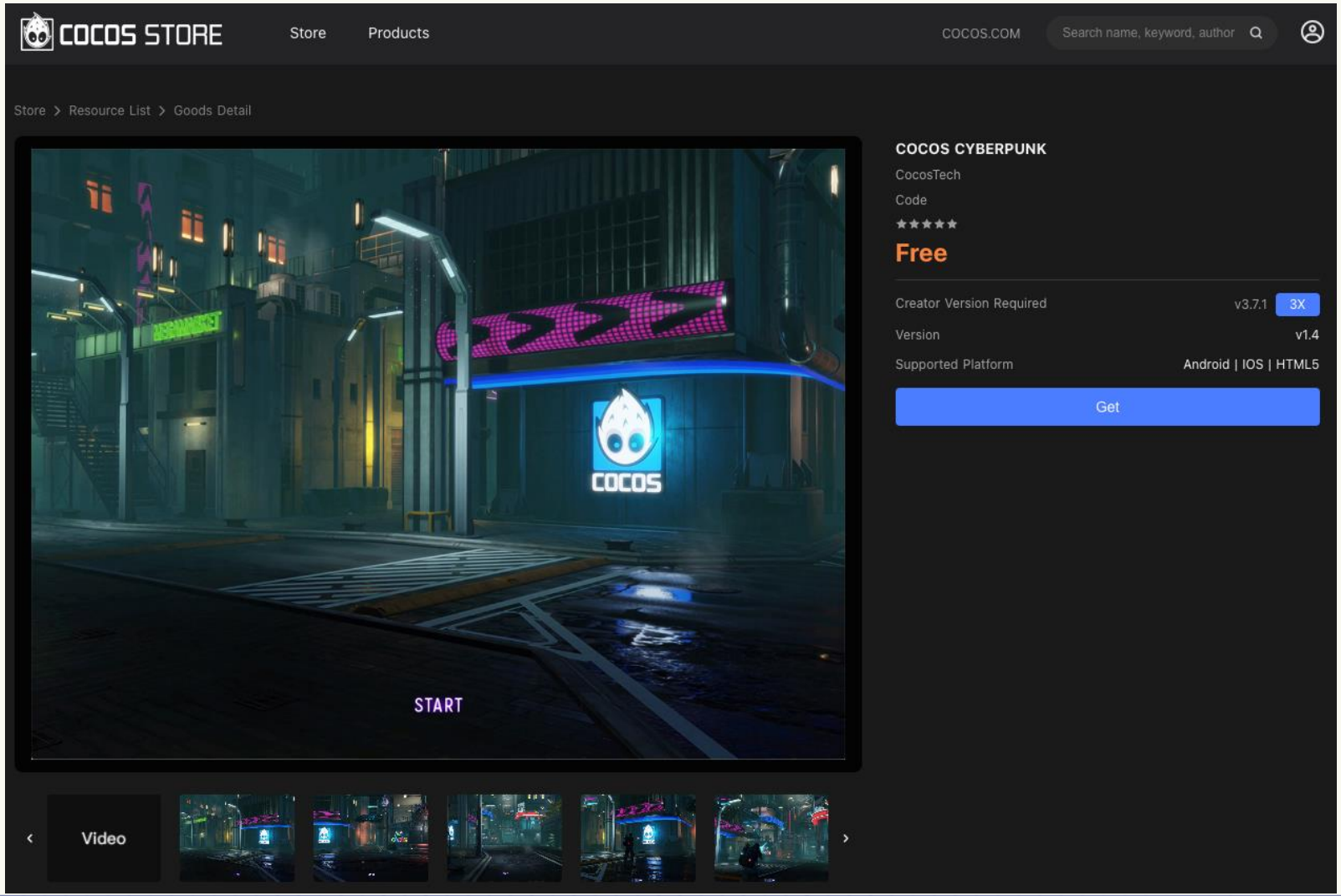


02.

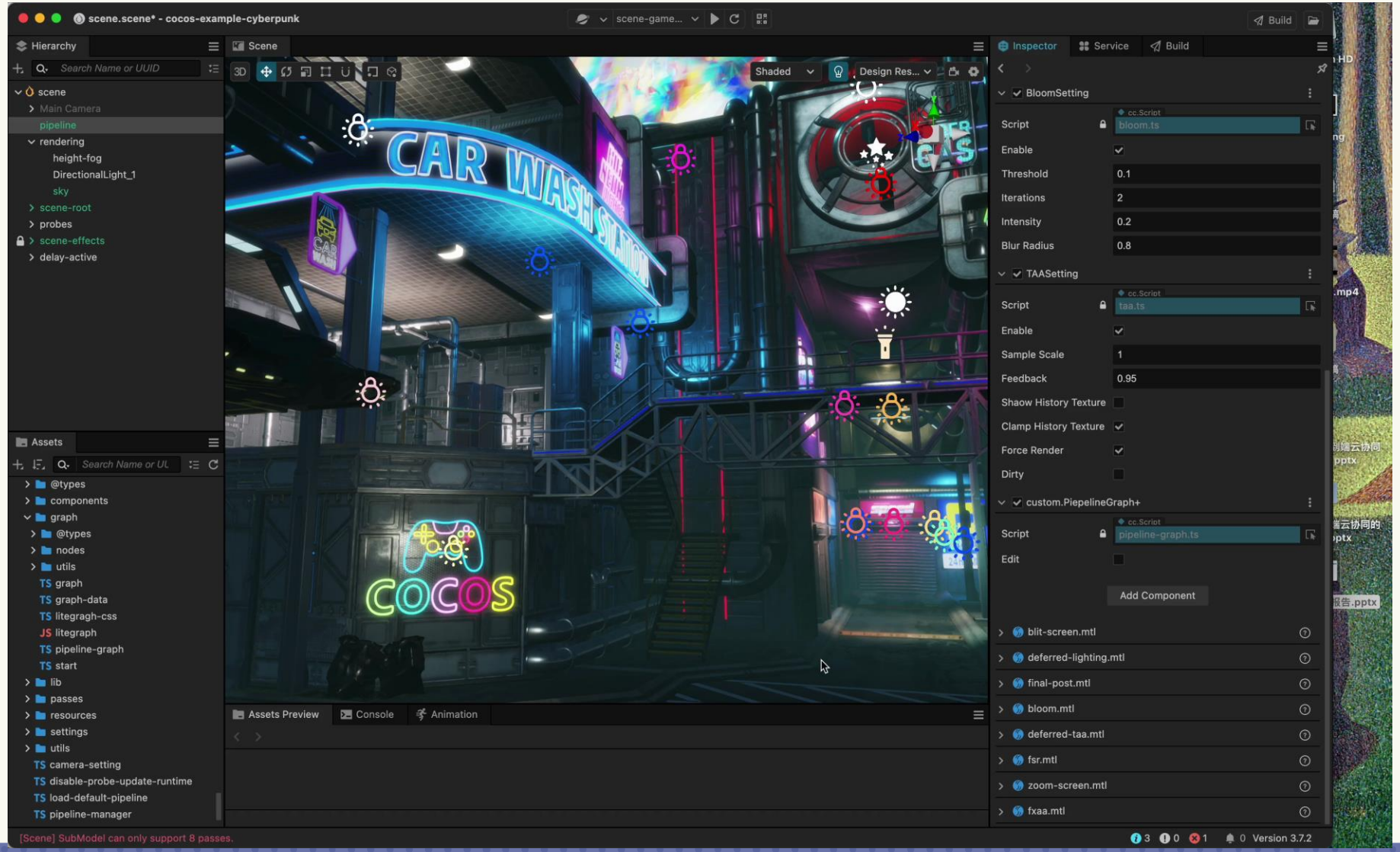
The Open Source Sample Project for Render Graph

Open Source Sample Project for Render Graph

Cyberpunk Demo published in Cocos Store



Render Graph Editing

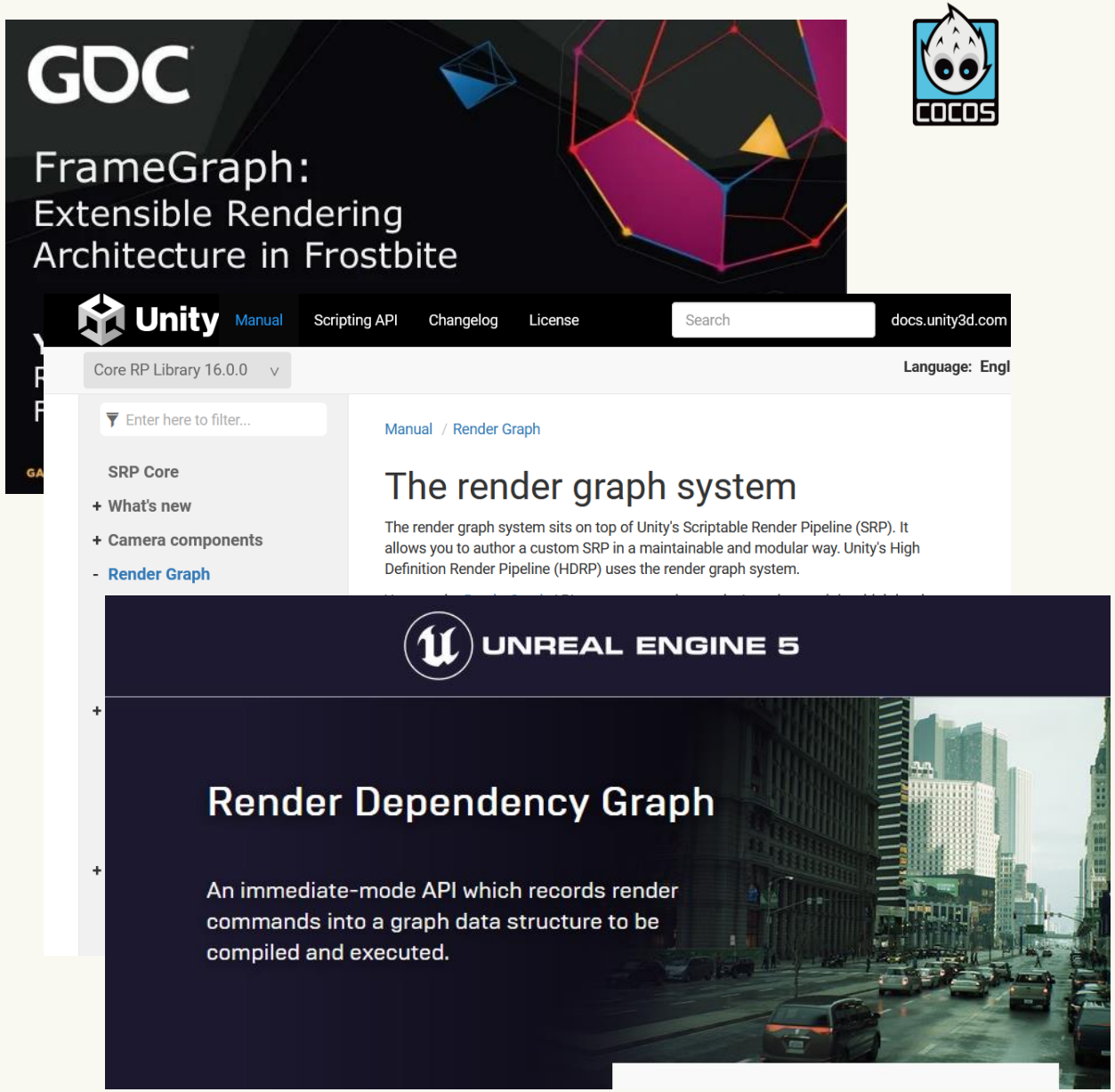


03.

Design Context of Render Graph

Frame Graph

- Build high-level knowledge of the entire frame
  - Simplify resource management
  - Simplify render pipeline configurations
  - Simplify async compute and resource barriers
- Allow self-contained and efficient rendering modules
- Visualize and debug complex rendering pipelines



Frame Graph vs Render Graph

Frame Graph

- Build high-level knowledge of the entire frame
  - Simplify resource management
  - Simplify render pipeline configurations
  - Simplify async compute and resource barriers
- Allow self-contained and efficient rendering modules
- Visualize and debug complex rendering pipelines

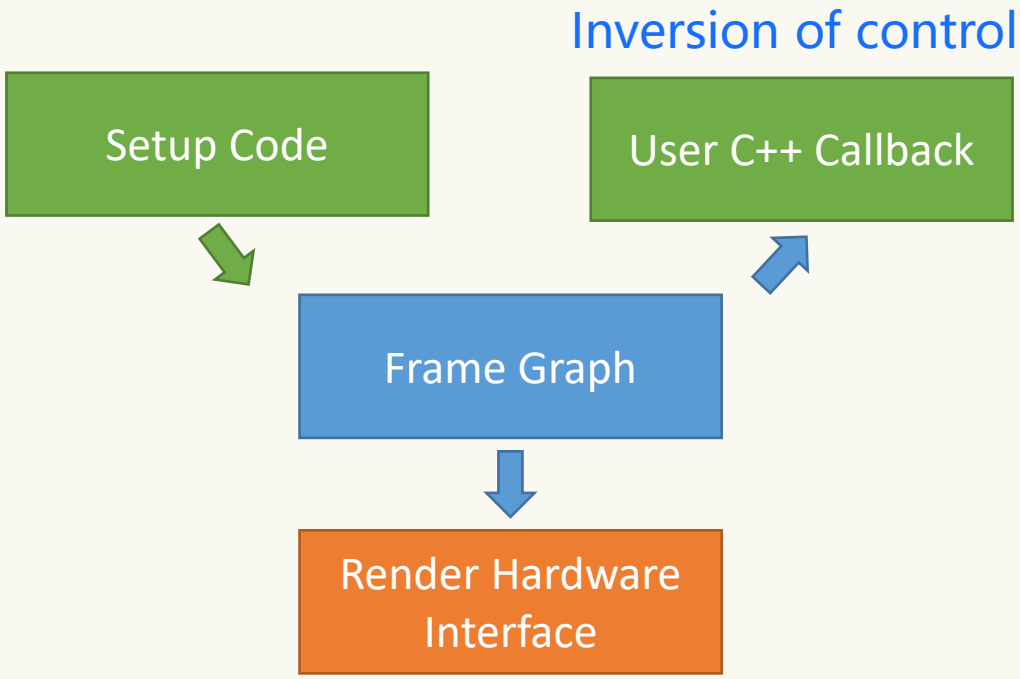
Render Graph (Data-oriented)

- Build high-level knowledge of the entire frame and scene
  - Full description of a rendering task
  - Simplify configurations with declarative programming
- Decouple pipeline setup and execution
  - Better testability
- Allow graph transformation and modification

Frame Graph vs Render Graph

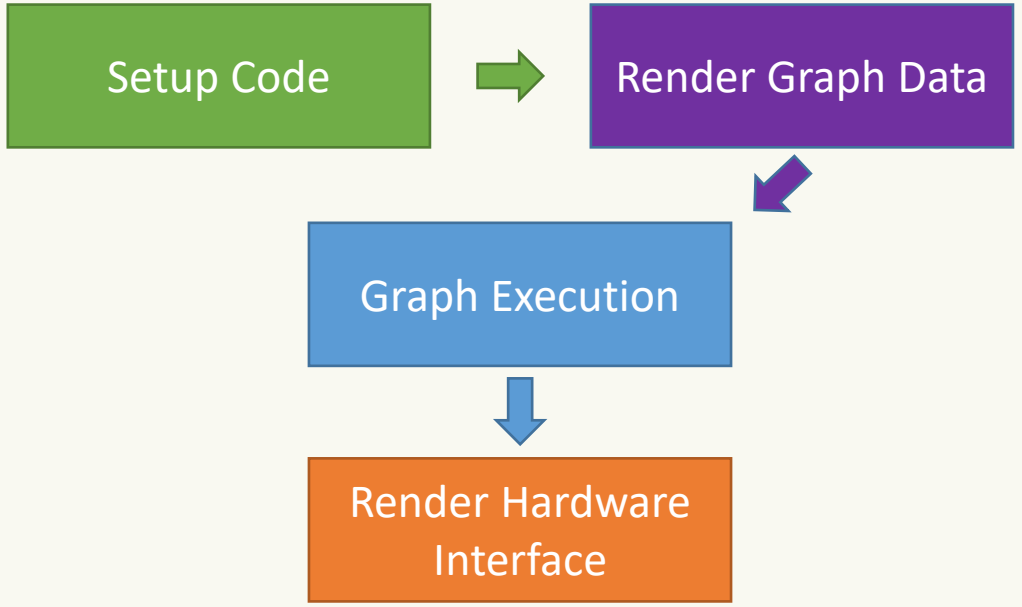
Frame Graph

- User write features as callbacks



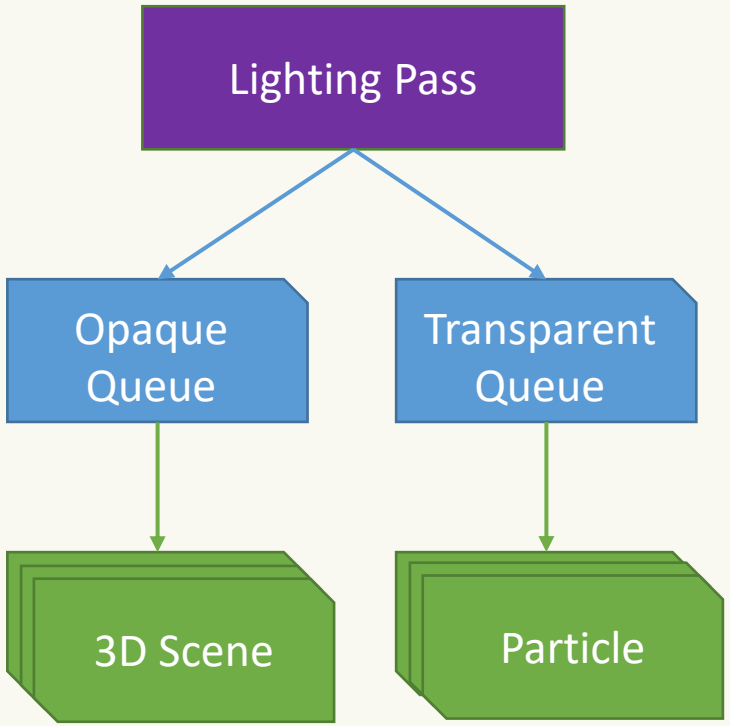
Render Graph (Data-oriented)

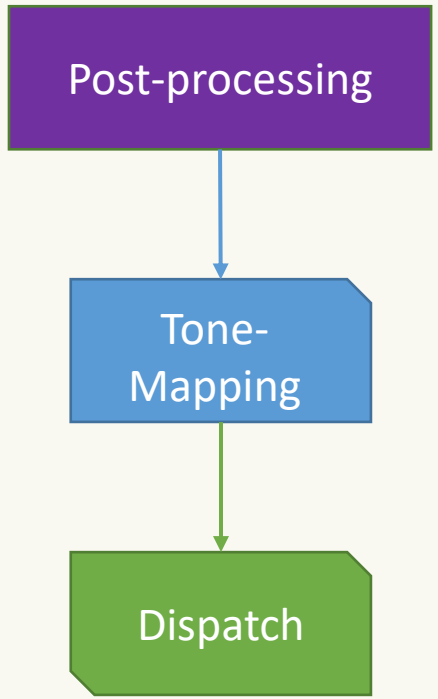
- User provides description



Graph data is layered

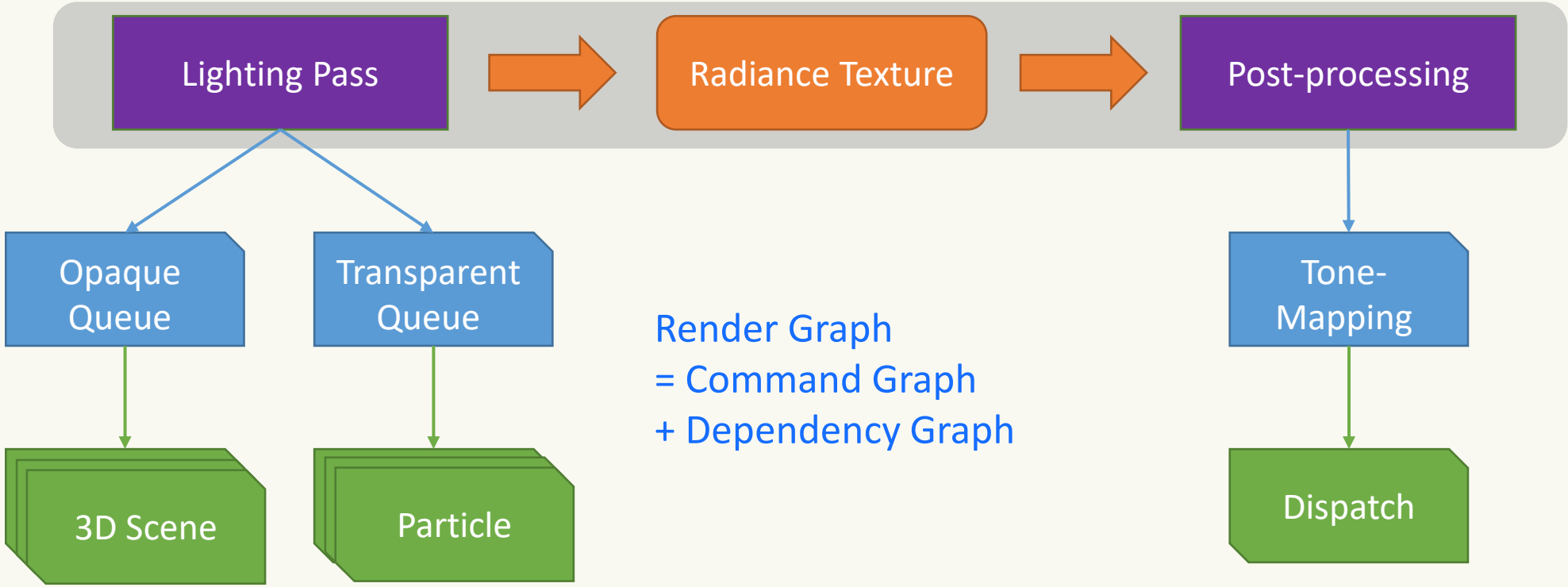
- Base Graph: Command Graph



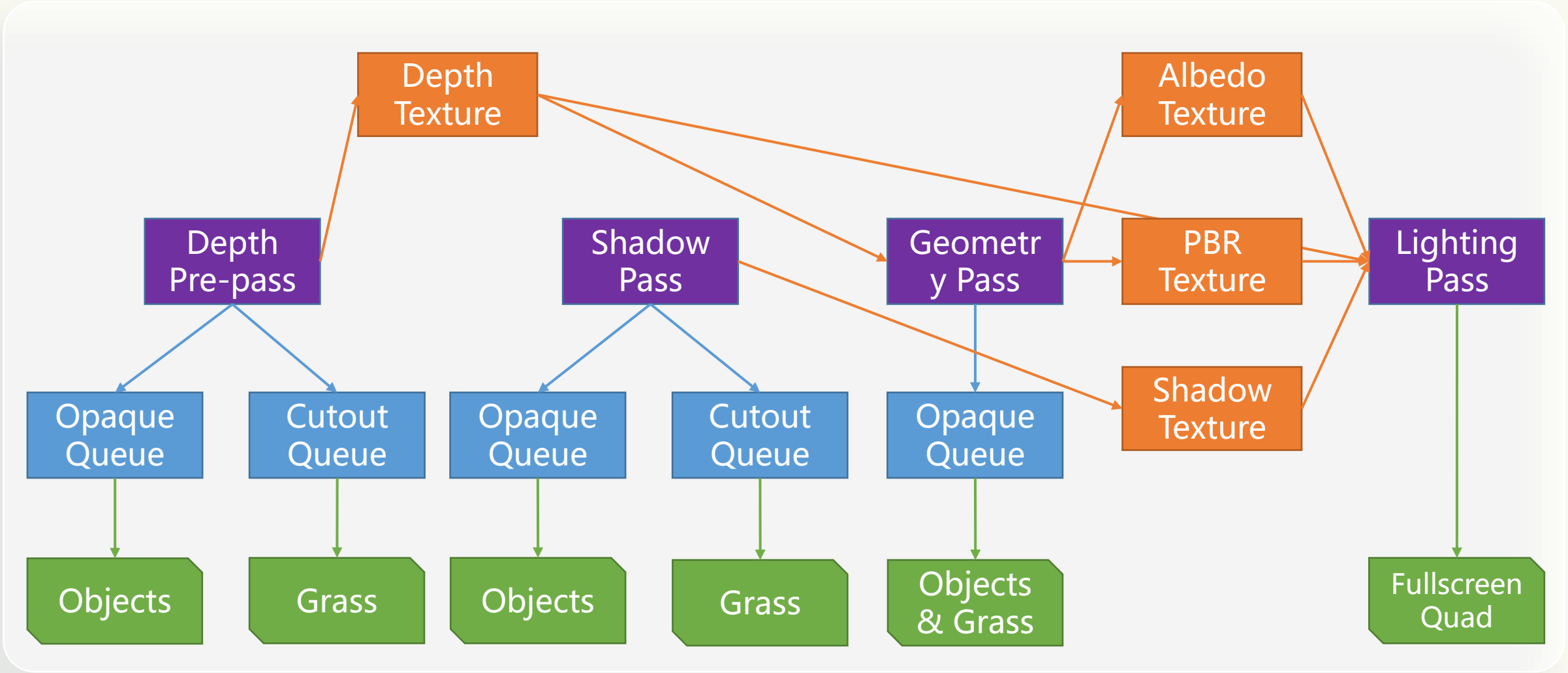


Graph data is layered

- Overlay Graph: Dependency Graph



Render Graph Example



Sample code: Graph setup

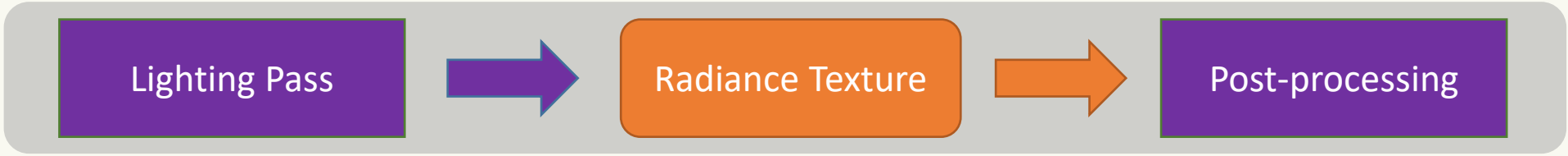
```
forwardPass.addRasterView(forwardPassRTName,
    new RasterView('_',
        AccessType.WRITE, AttachmentType.RENDER_TARGET,
        cameraRenderTargetLoadOp,
        StoreOp.STORE,
        getClearFlags(AttachmentType.RENDER_TARGET, camera.clearFlag, cameraRenderTargetLoadOp),
        new Color(camera.clearColor.x, camera.clearColor.y, camera.clearColor.z, camera.clearColor.w)));
forwardPass.addRasterView(forwardPassDSName,
    new RasterView('_',
        AccessType.WRITE, AttachmentType.DEPTH_STENCIL,
        cameraDepthStencilLoadOp,
        StoreOp.STORE,
        getClearFlags(AttachmentType.DEPTH_STENCIL, camera.clearFlag, cameraDepthStencilLoadOp),
        new Color(camera.clearDepth, camera.clearStencil, 0, 0)));

forwardPass
    .addQueue(QueueHint.RENDER_OPAQUE)
    .addSceneOfCamera(camera, new LightInfo(),
        SceneFlags.OPAQUE_OBJECT
        | SceneFlags.PLANAR_SHADOW
        | SceneFlags.CUTOUT_OBJECT
        | SceneFlags.DEFAULT_LIGHTING
        | SceneFlags.DRAW_INSTANCING);
forwardPass
    .addQueue(QueueHint.RENDER_TRANSPARENT)
    .addSceneOfCamera(camera, new LightInfo(),
        SceneFlags.TRANSPARENT_OBJECT
        | SceneFlags.GEOMETRY);
forwardPass
    .addQueue(QueueHint.RENDER_TRANSPARENT)
    .addSceneOfCamera(camera, new LightInfo(),
        SceneFlags.UI
        | SceneFlags.PROFILER);
```

Graph data is inspectable

- Compiler/Analyzer is easy to write
  - Reflection is not needed

User Input

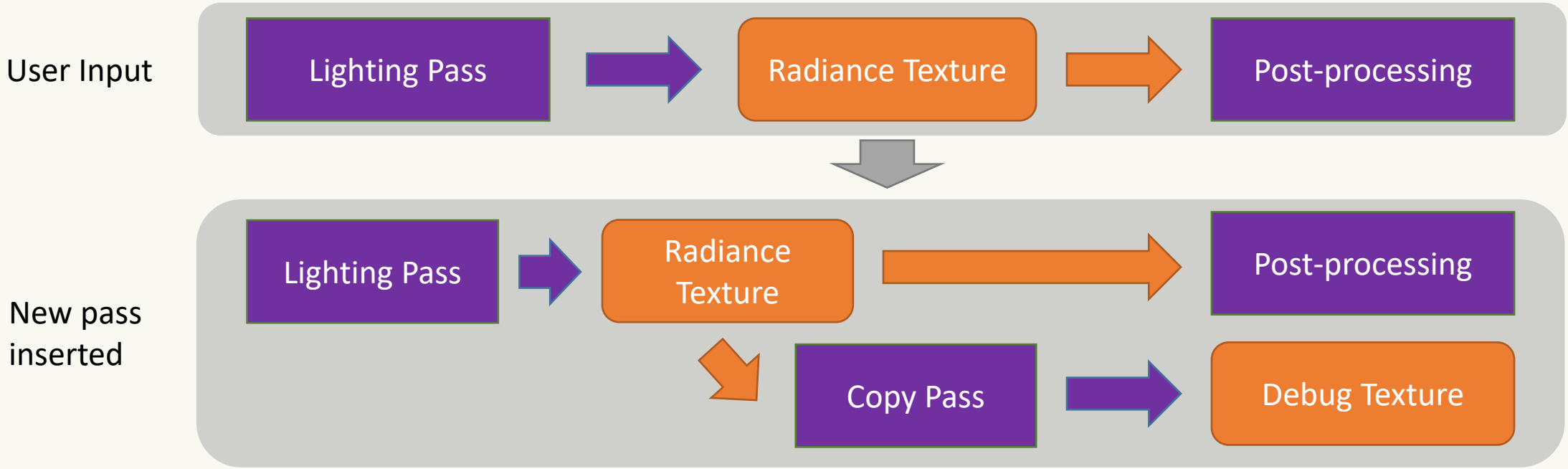


Deduced execution plan

Schedule	Barrier Before	Barrier After	Schedule
Direct Queue	Sync: Render Target	Sync: Compute	Direct Queue
	Access: Render Target	Access: Shader Resource	Wait for Lighting Pass
	Layout: Render Target	Layout: Shader Resource	
	Sub-resource: 0		

Graph data is mutable

- Engine can modify render graph
  - User code is the same
  - Execution code is the same



Descriptor Layout Optimization

- Simplify shader management
  - Render Graph need descriptor layout
  - Hand-written layout is error prone

A	Layout 0
Set 0	Texture2D Lightmap
Set 1	
Set 2	Texture2D Main
Set 3	

B	Layout 1
Set 0	Texture2D LUT
Set 1	
Set 2	Texture2D BaseColor Texture2D Normal
Set 3	

- Render Pass
  - Shader A
    - Bind Set 0
    - Bind Set 2
    - Draw
  - Shader B
    - Bind Set 0
    - Bind Set 2
    - Draw

Merge layout

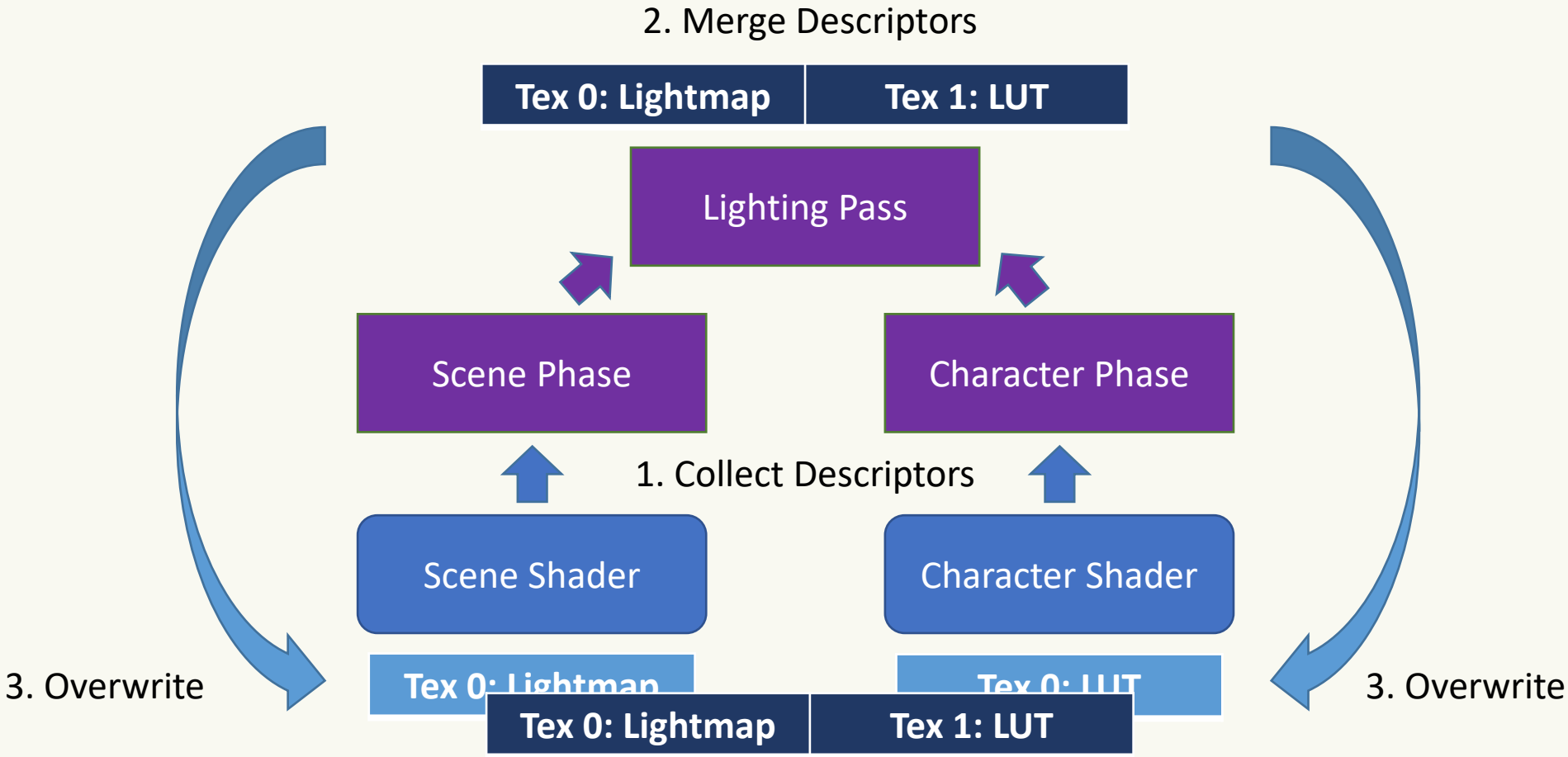
A	Layout 0
Set 0	Texture2D Lightmap Texture2D LUT
Set 1	
Set 2	Texture2D Main Texture2D [Empty]
Set 3	

B	Layout 0
Set 0	Texture2D Lightmap Texture2D LUT
Set 1	
Set 2	Texture2D BaseColor Texture2D Normal
Set 3	

- Render Pass
  - Bind Set 0
  - Shader A
    - Bind Set 2
    - Draw
  - Shader B
    - Bind Set 2
    - Draw

Fewer state changes

Descriptor Layout Graph



04.

Core Design Explained

Example: data structure optimization

- Array of Structure

Vertex	v0	v1	v2
Out Edges	…	…	…
In Edges			
Type			
Name			
Data			

- Profile and decide implementation
- Should use same access interface
  - get(property, g, v)

- Structure of Array

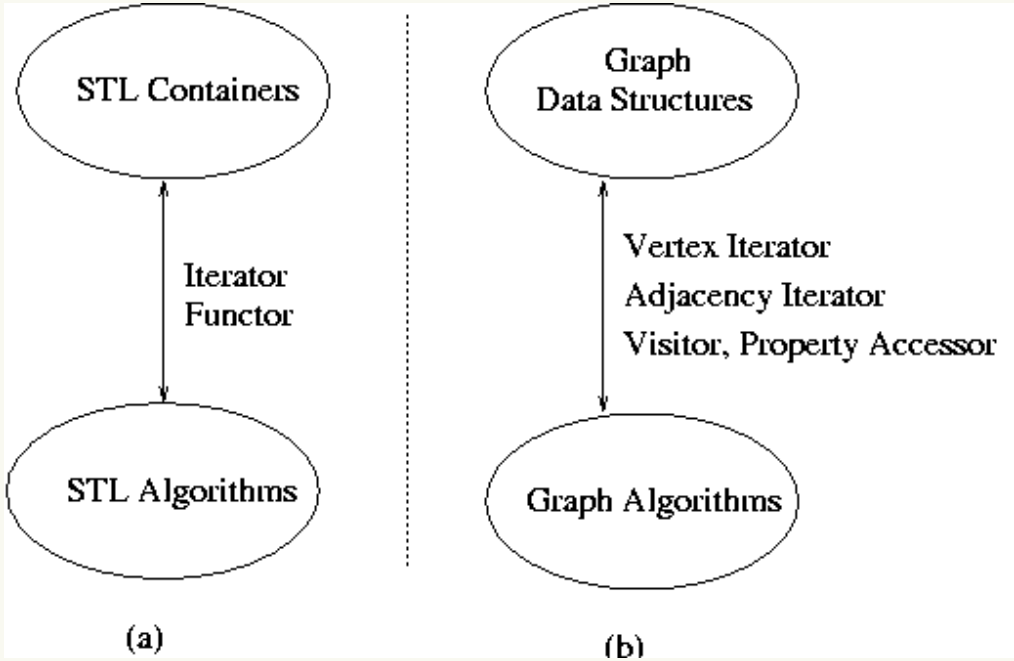
Vertex	v0	v1	v2
Out Edges	…	…	…
In Edges			
Type			

Property	v0	v1	v2
Name	…	…	…

Property	v0	v1	v2
Data	…	…	…

Generic Graph Interface

- Based on boost.graph
  - Decouple graph data structure and graph algorithms
  - Zero-overhead abstraction
- Many existing graph algorithms
  - Reduce development cost
- There are a lot of graphs in game engine!
  - Render Graph, Scene Graph, Shader Graph, Behavior Tree, Pathfinding, etc.
  - All benefit from a generic graph interface



Generic graph interface

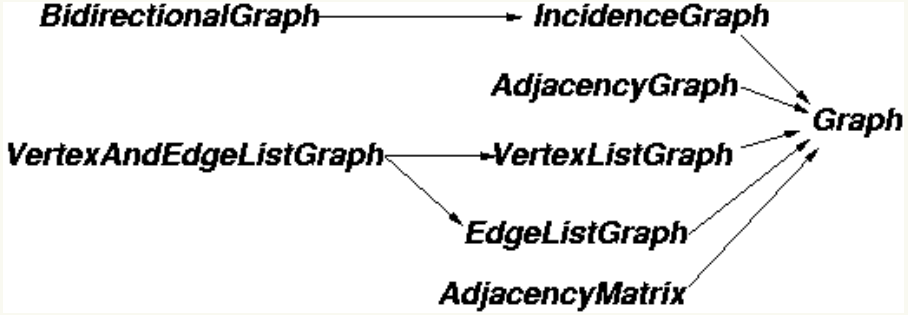
- Generic algorithms built on graph concepts

Bidirectional Graph
Depth First Search
Topological Sort
Graph Coloring

Property Graph
Getter
Setter
Property Map

Addressable Graph
Get Parent
Get Child
Lookup Path

- Reduce implementation cost
  - O(M x N) -> O(M + N), where
    - M is number of graph types
    - N is number of algorithms



Implementation details

- **Code generator**
  - Written in C++
  - Register types with DSL
  - Generate graph implementations
    - C++ and Typescript
- **PROS**
  - Support more features than generic implementation
  - No template meta-programming is required
  - Generated code is easier to read
- **CONS**
  - Introduced another layer of tool
  - A lot of type registration

```
PMR_GRAPH(RenderGraph, _, _, .mFlags = NO_COPY) {
    NAMED_GRAPH(Name_);
    REFERENCE_GRAPH();

    COMPONENT_GRAPH(
        (Name_,    ccstd::pmr::string, mNames)
        (Layout_,  ccstd::pmr::string, mLayoutNodes)
        (Data_,    RenderData,         mData)
        (Valid_,   bool,               mValid)
    );

    POLYMORPHIC_GRAPH(
        (RasterPass_,     RasterPass,                    mRasterPasses)
        (RasterSubpass_,  RasterSubpass,                 mRasterSubpasses)
        (ComputeSubpass_, ComputeSubpass,                mComputeSubpasses)
        (Compute_,        ComputePass,                   mComputePasses)
        (Copy_,           CopyPass,                      mCopyPasses)
        (Move_,           MovePass,                      mMovePasses)
        (Raytrace_,       RaytracePass,                  mRaytracePasses)
        (Queue_,          RenderQueue,                   mRenderQueues)
        (Scene_,          SceneData,                     mScenes)
        (Blit_,           Blit,                          mBlits)
        (Dispatch_,       Dispatch,                      mDispatches)
        (Clear_,          ccstd::pmr::vector<ClearView>, mClearViews)
        (Viewport_,       gfx::Viewport,                 mViewports)
    );
}
```

Graph concepts references

- Supports C++ and Typescript
- Common concepts
  - Graph
  - Incidence Graph
  - Bidirectional Graph
  - Adjacency Graph
  - Vertex List Graph
  - Edge List Graph

```typescript
export interface Graph {
    readonly directed_category: directional;
    readonly edge_parallel_category: parallel;
    readonly traversal_category: traversal;

    nullVertex (): vertex_descriptor | null;
}

export interface IncidenceGraph extends Graph {
    edge (u: vertex_descriptor, v: vertex_descriptor): boolean;
    source (e: edge_descriptor): vertex_descriptor;
    target (e: edge_descriptor): vertex_descriptor;
    outEdges (v: vertex_descriptor): out_edge_iterator;
    outDegree (v: vertex_descriptor): number;
}

export interface BidirectionalGraph extends IncidenceGraph {
    inEdges (v: vertex_descriptor): in_edge_iterator;
    inDegree (v: vertex_descriptor): number;
    degree (v: vertex_descriptor) : number;
}

export interface VertexListGraph extends Graph {
    vertices (): IterableIterator<vertex_descriptor>;
    numVertices (): number;
}

export interface EdgeListGraph extends Graph {
    edges (): IterableIterator<edge_descriptor>;
    numEdges (): number;
    source (e: edge_descriptor): vertex_descriptor;
    target (e: edge_descriptor): vertex_descriptor;
}

export interface MutableGraph extends Graph {
    addVertex (...args): vertex_descriptor;
    clearVertex (v: vertex_descriptor): void;
    removeVertex (v: vertex_descriptor): void;
    addEdge (u: vertex_descriptor, v: vertex_descriptor,
        p?: unknown): edge_descriptor | null;
    removeEdges (u: vertex_descriptor, v: vertex_descriptor): void;
    removeEdge (e: edge_descriptor): void;
}
```

Graph concepts references

- Component Graph
- Named Graph
- Tree
- Parent Graph
  - Requires Named Graph
  - Requires Tree
- Addressable Graph
  - Requires Parent Graph
- Polymorphic Graph
- UUID Graph

```
export interface Tree extends Graph {
    reference (u: vertex_descriptor, v: vertex_descriptor): boolean;
    parent (e: reference_descriptor): vertex_descriptor;
    child (e: reference_descriptor): vertex_descriptor;
    parents (v: vertex_descriptor): parent_iterator;
    children (v: vertex_descriptor): child_iterator;
    numParents (v: vertex_descriptor): number;
    numChildren (v: vertex_descriptor): number;
    getParent (v: vertex_descriptor): vertex_descriptor | null;
    isAncestor (ancestor: vertex_descriptor,
        descendent: vertex_descriptor): boolean;
}

export interface MutableTree extends Tree {
    addReference (u: vertex_descriptor, v: vertex_descriptor,
        p?: unknown): reference_descriptor | null;
    removeReference (e: reference_descriptor): void;
    removeReferences (u: vertex_descriptor, v: vertex_descriptor): void;
}

export interface ParentGraph extends Tree, NamedGraph {
    locateChild (v: vertex_descriptor | null,
        name: string): vertex_descriptor | null;
}

export interface AddressableGraph extends ParentGraph {
    addressable (absPath: string): boolean;
    locate (absPath: string): vertex_descriptor | null;
    locateRelative (path: string,
        start?: vertex_descriptor | null): vertex_descriptor | null;
    path (v: vertex_descriptor) : string;
}
```

Render Graph implementation

RenderGraph

- Bidirectional Graph
- Vertex List Graph
- Edge List Graph
- Component Graph
- Named Graph
- Tree
- Polymorphic Graph

LayoutGraph

- Bidirectional Graph
- Vertex List Graph
- Component Graph
- Named Graph
- Parent Graph
- Addressable Graph
- Polymorphic Graph

Thanks !
Questions ?

github.com/cocos/cocos-engine/
www.cocos.com/en/creator-download
store.cocos.com/app/en/detail/4543
@CocosEngine