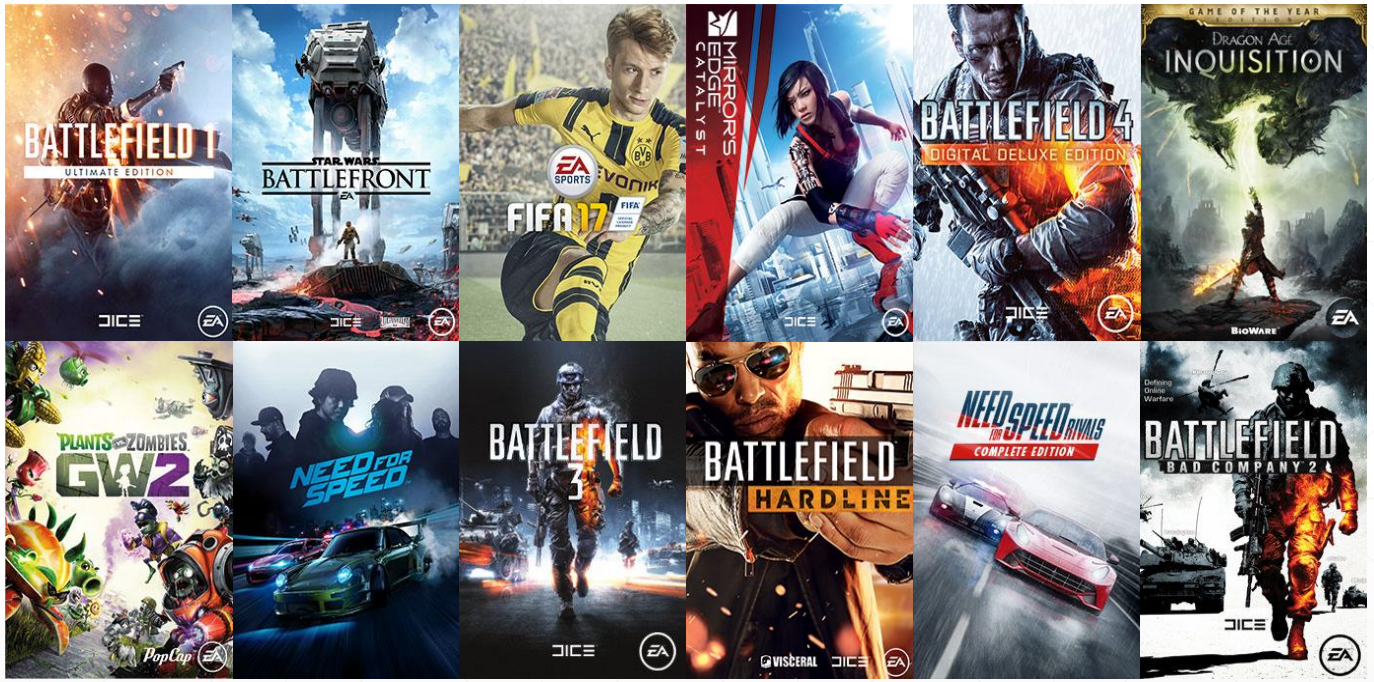
Dice (EA) 工作室游戏开发技术概览

Dice 如何改造引擎适应现代技术发展

"Move to One Engine"

Frostbite



- **FIFA** (FIFA 17/18)
- **Mass Effect** (Mass Effect: Andromeda)
- **Battlefield** (Battlefield 1 - 2016)
- **Star Wars** (Star Wars Battlefront)
- **Dragon Age** (Dragon Age: Origins)
- **Mirror's Edge** (Mirror's Edge Catalyst - 2016)
- **Need for Speed** (NEED FOR SPEED 2017: PAYING IT BACK)

EA is moving its games, including Mass Effect and FIFA, onto a single graphics engine.
Frostbite has evolved to become the cornerstone of these titles.

目录

宽泛 & 系统

明确 & 具体

A. 移动化

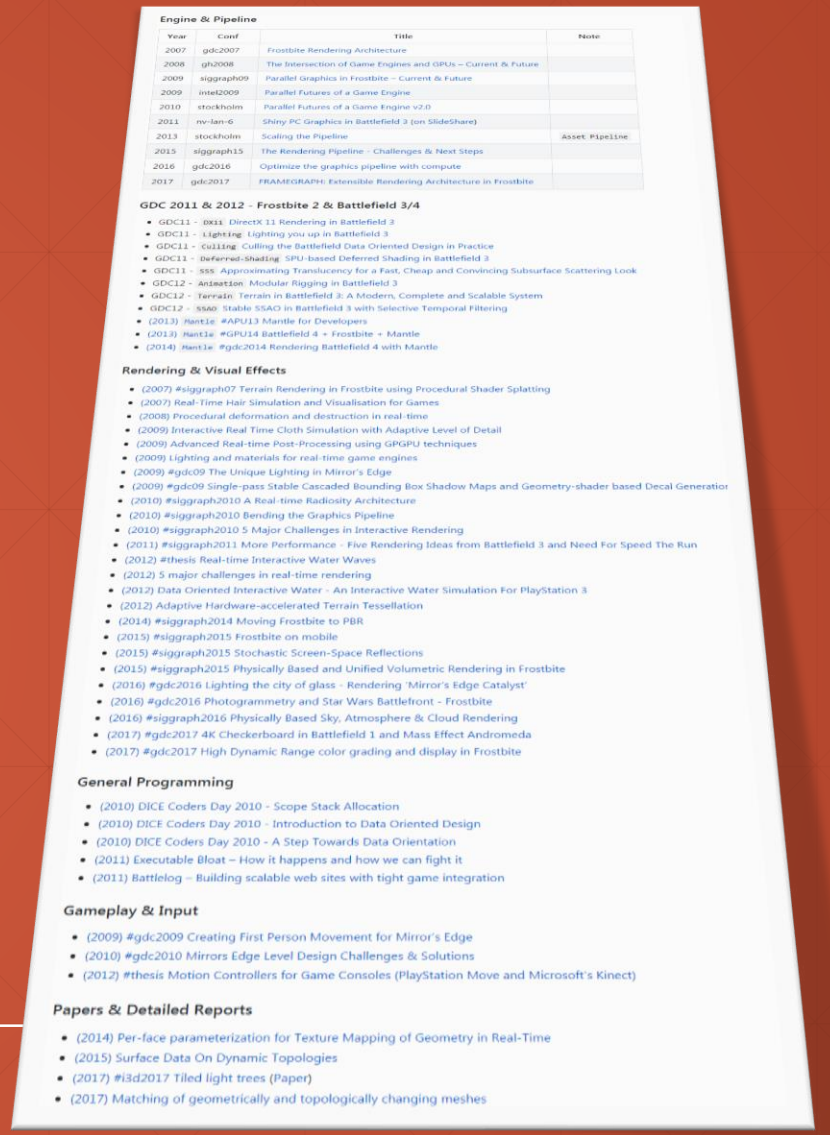
Frostbite 面向移动平台的改造

B. 代码实现

围绕数据的改造（Data-Oriented）

C. 架构改造

FrameGraph 可扩展的渲染架构



Engine & Pipeline

Year	Conf	Title	Note
2007	gdc2007	Frostbite Rendering Architecture	
2008	gh2008	The Intersection of Game Engines and GPUs – Current & Future	
2009	siggraph09	Parallel Graphics in Frostbite – Current & Future	
2009	intel2009	Parallel Futures of a Game Engine	
2010	stockholm	Parallel Futures of a Game Engine v2.0	
2011	nv-lan-6	Shiny PC Graphics in Battlefield 3 (on SlideShare)	
2013	stockholm	Scaling the Pipeline	Asset Pipeline
2015	siggraph15	The Rendering Pipeline - Challenges & Next Steps	
2016	gdc2016	Optimize the graphics pipeline with compute	
2017	gdc2017	FRAMEGRAPH: Extensible Rendering Architecture in Frostbite	

GDC 2011 & 2012 - Frostbite 2 & Battlefield 3/4

- GDC11 - `DX11` DirectX 11 Rendering in Battlefield 3
- GDC11 - `Lighting` Lighting you up in Battlefield 3
- GDC11 - `Culling` Culling the Battlefield Data Oriented Design in Practice
- GDC11 - `Deferred-Shading` SPU-based Deferred Shading in Battlefield 3
- GDC11 - `SSS` Approximating Translucency for a Fast, Cheap and Convincing Subsurface Scattering Look
- GDC12 - `Animation` Modular Rigging in Battlefield 3
- GDC12 - `Terrain` Terrain in Battlefield 3: A Modern, Complete and Scalable System
- GDC12 - `SSAO` Stable SSAO in Battlefield 3 with Selective Temporal Filtering
- (2013) `Mantle` #APU13 Mantle for Developers
- (2013) `Mantle` #GPU14 Battlefield 4 + Frostbite + Mantle
- (2014) `Mantle` #gdc2014 Rendering Battlefield 4 with Mantle

Rendering & Visual Effects

- (2007) #siggraph07 Terrain Rendering in Frostbite using Procedural Shader Splatting
- (2007) Real-Time Hair Simulation and Visualisation for Games
- (2008) Procedural deformation and destruction in real-time
- (2009) Interactive Real Time Cloth Simulation with Adaptive Level of Detail
- (2009) Advanced Real-time Post-Processing using GPGPU techniques
- (2009) Lighting and materials for real-time game engines
- (2009) #gdc09 The Unique Lighting in Mirror's Edge
- (2009) #gdc09 Single-pass Stable Cascaded Bounding Box Shadow Maps and Geometry-shader based Decal Generation
- (2010) #siggraph2010 A Real-time Radiosity Architecture
- (2010) #siggraph2010 Bending the Graphics Pipeline
- (2010) #siggraph2010 5 Major Challenges in Interactive Rendering
- (2011) #siggraph2011 More Performance - Five Rendering Ideas from Battlefield 3 and Need For Speed The Run
- (2012) #thesis Real-time Interactive Water Waves
- (2012) 5 major challenges in real-time rendering
- (2012) Data Oriented Interactive Water - An Interactive Water Simulation For PlayStation 3
- (2012) Adaptive Hardware-accelerated Terrain Tessellation
- (2014) #siggraph2014 Moving Frostbite to PBR
- (2015) #siggraph2015 Frostbite on mobile
- (2015) #siggraph2015 Stochastic Screen-Space Reflections
- (2015) #siggraph2015 Physically Based and Unified Volumetric Rendering in Frostbite
- (2016) #gdc2016 Lighting the city of glass - Rendering 'Mirror's Edge Catalyst'
- (2016) #gdc2016 Photogrammetry and Star Wars Battlefront - Frostbite
- (2016) #siggraph2016 Physically Based Sky, Atmosphere & Cloud Rendering
- (2017) #gdc2017 4K Checkerboard in Battlefield 1 and Mass Effect Andromeda
- (2017) #gdc2017 High Dynamic Range color grading and display in Frostbite

General Programming

- (2010) DICE Coders Day 2010 - Scope Stack Allocation
- (2010) DICE Coders Day 2010 - Introduction to Data Oriented Design
- (2010) DICE Coders Day 2010 - A Step Towards Data Orientation
- (2011) Executable Bloat – How it happens and how we can fight it
- (2011) Battlelog – Building scalable web sites with tight game integration

Gameplay & Input

- (2009) #gdc2009 Creating First Person Movement for Mirror's Edge
- (2010) #gdc2010 Mirrors Edge Level Design Challenges & Solutions
- (2012) #thesis Motion Controllers for Game Consoles (PlayStation Move and Microsoft's Kinect)

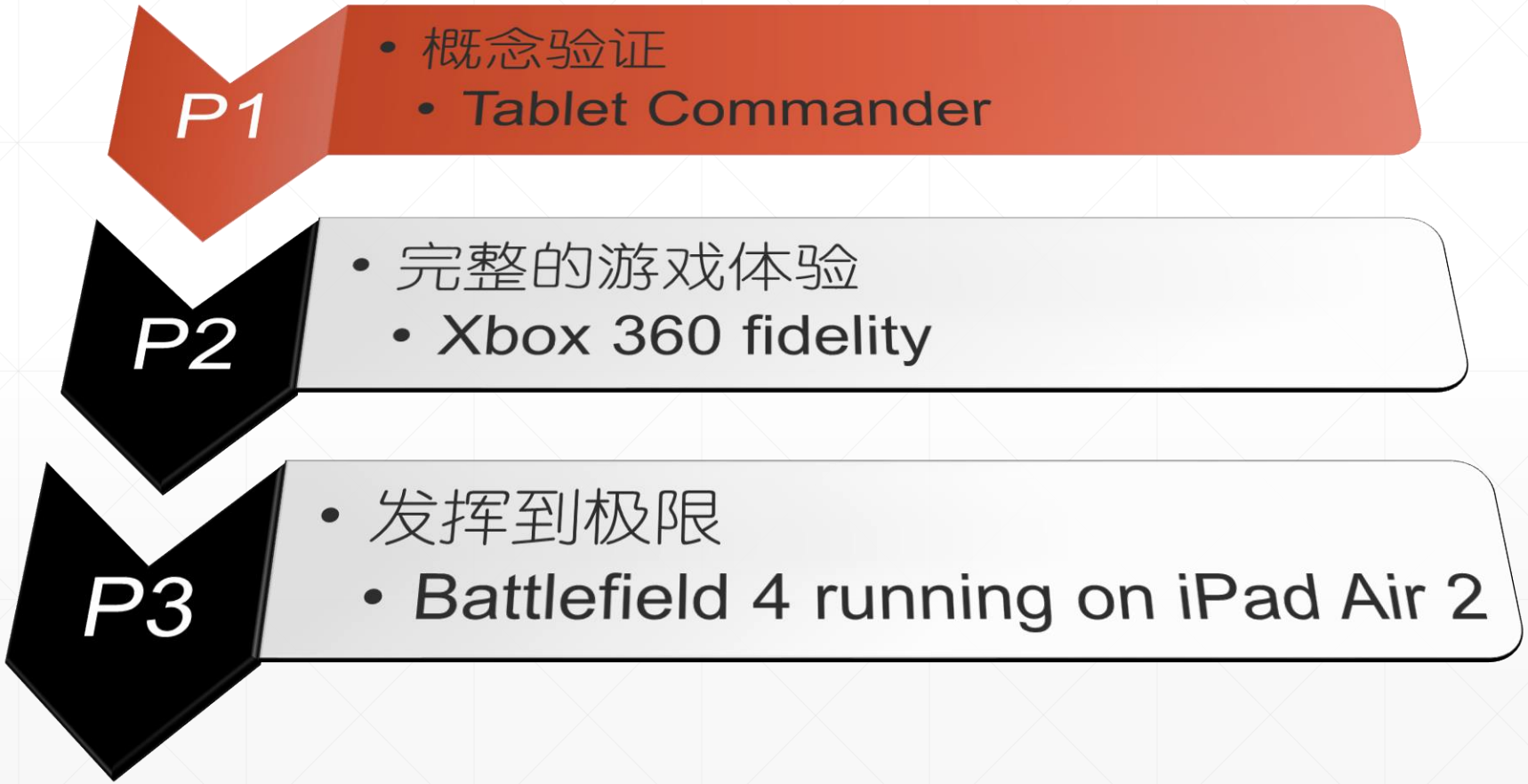
Papers & Detailed Reports

- (2014) Per-face parameterization for Texture Mapping of Geometry in Real-Time
- (2015) Surface Data On Dynamic Topologies
- (2017) #i3d2017 Tiled light trees (Paper)
- (2017) Matching of geometrically and topologically changing meshes

“55 份材料” → “三个要点”

Check-out the full-list here: https://github.com/mc-gulu/dev-awesomenesses/blob/master/awesome-frostbite-engine.md

移动化 (1/3)


P1
• 概念验证
• Tablet Commander
P2
• 完整的游戏体验
• Xbox 360 fidelity
P3
• 发挥到极限
• Battlefield 4 running on iPad Air 2


MENU
A B D C E
US 300
00:45:23
CN 99
SCOREBOARD
Feffalump [G36C] LeCrap
Anpanfisk [G36C] IICarpentero
US
SCAN UAV
EMP UAV
EVAC ORDER
HIGH VALUE TARGET
B
VEHICLE SCAN
C
CRUISE MISSILE
D
INFANTRY SCAN
VEHICLE DROP
RAPID DEPLOY
SUPPLY DROP
PROMOTE SQUAD
ALPHA
BETA
CHARLIE
DELTA
ECHO
FOXTROT
GOLF
50
TheHelgon
Manualmartin
OhPlz
Megalodon
CRUISE MISSILE LAUNCHED +50
SCAN BONUS +25
ORDER ACCEPTED +10
Manualmartin

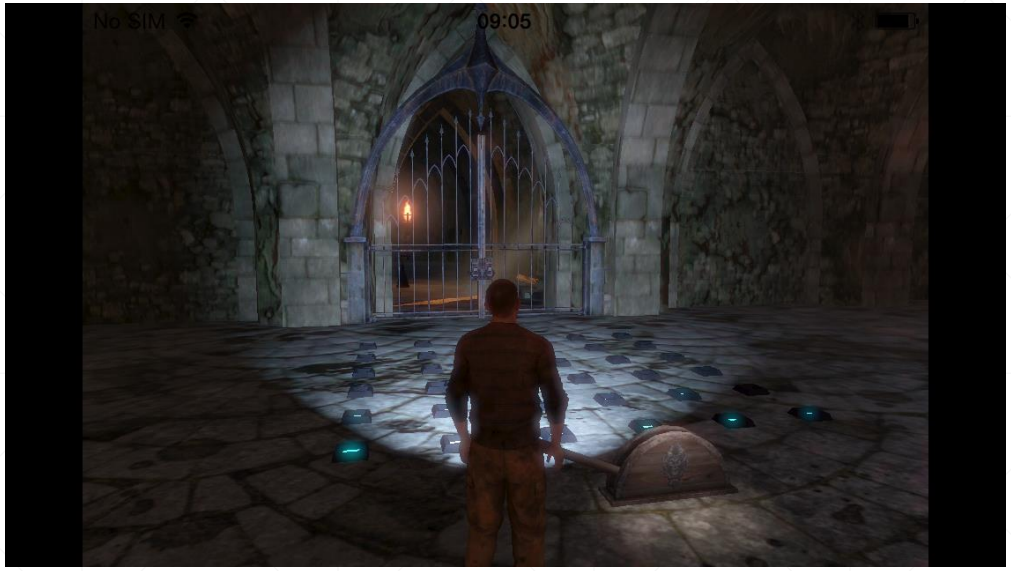
P1 - 概念验证：Tablet Commander

- 指挥官模式 (The commander mode) 在此前的 Battlefield 系列中是一个玩法，在 Battlefield 4 中分拆成了移动设备上的一个独立的应用。
- 多个好处：
  - 仅接触并改造整个代码库中的一小部分
  - 考察引擎的可移植性
  - 考察PC/Console与移动平台互通性

移动化 (2/3)


p1
• 概念验证
• Tablet Commander
p2
• 完整的游戏体验
• Xbox 360 fidelity
P3
• 发挥到极限
• Battlefield 4 running on iPad Air 2

- OpenGL ES 2.0 + extensions → OpenGL ES 3.0



- Shader Works
  - 对PP等手写的 shader，手动移植并做平台优化
  - 对基于节点生成的 shader，添加 GLSL 生成支持
  - 使用 **GLSL-optimizer** 优化尺寸和执行效率 (Aras@Unity)

P2 - 完整的游戏体验

- 阶段性目标
  - 相对完整的体验，达到上一代主机 Xbox 360 水准
- 对伸缩性的考验
  - “we wanted to see if the dynamic nature of the Frostbite engine could play well on mobile and as the engine already scaled from 360 to gen4 console and PC quality, we predicted this should be possible.”

Reached steady 30 fps with SSAO, DOF, Radiosity, cascaded shadow maps, HDR, color grading, FXAA and lots of draw calls on a stock iPad Air 1

Based on Metal

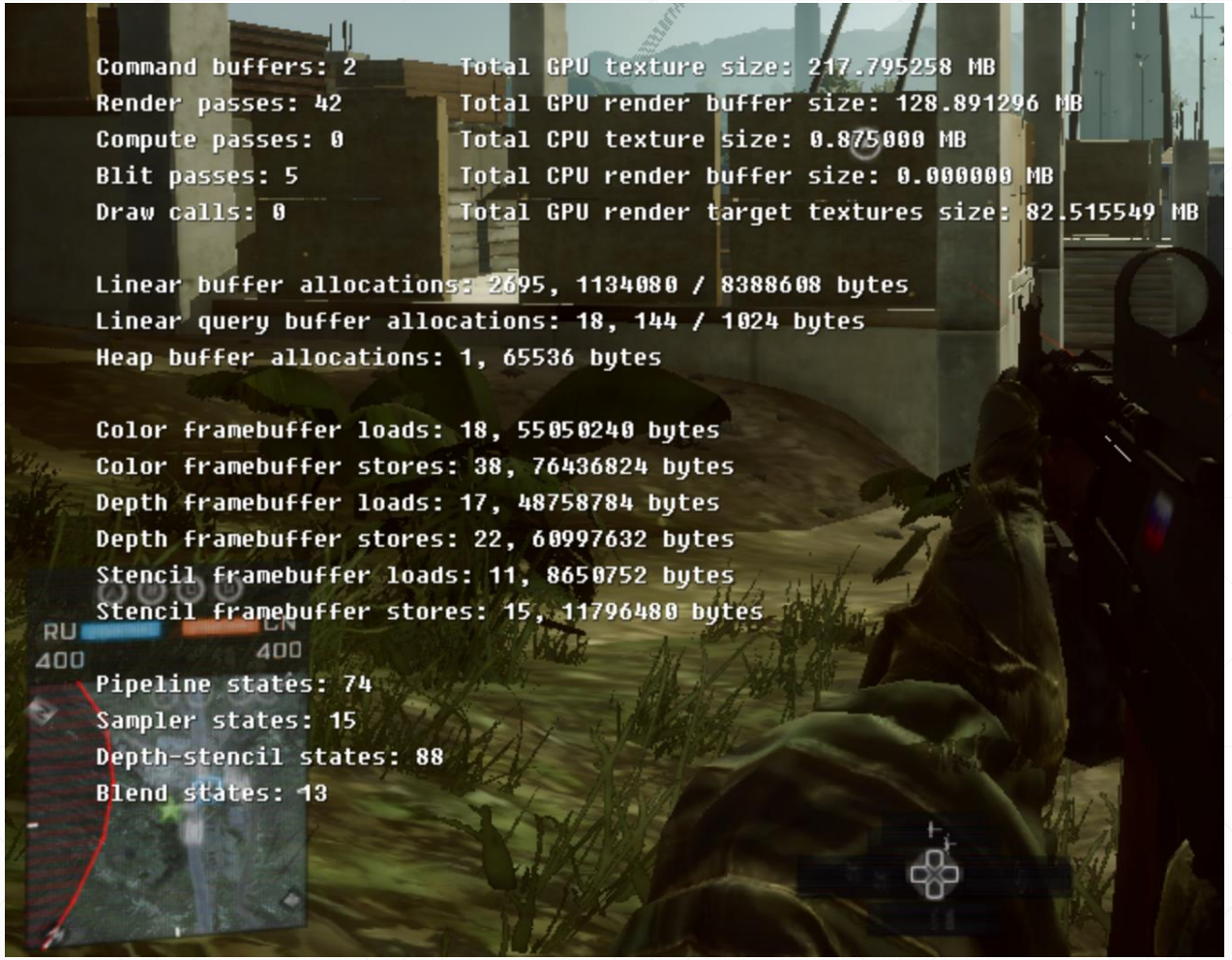
WWDC

- 首个基于 Metal 的版本
  - 使用 glsl-optimizer 生成 Metal 代码 (merged officially)
- 性能:
  - 5X lower overhead (与 OpenGL 相比)
  - 最少量的 Alpha tested 对象
  - Instancing 的效果不再那么明显

移动化 (3/3)


p1
• 概念验证
• Tablet Commander
p2
• 完整的游戏体验
• Xbox 360 fidelity
P3
• 发挥到极限
• Battlefield 4 running on iPad Air 2

挑战 1：内存问题



- fine grained statistics for gpu textures / buffers
- the engine have tags for all the resources
- printing all resources by **size** and **usage frequency**
  - to easily see what we should focus on and could get rid of
- be able to profile as much as possible **with their own cross platform code**

挑战 2：Shader works

- a lot of custom shaders were written in pure HLSL
  - simply too much work to port manually
  - it is a pain to maintain multiple versions of shaders
  - mobile shader languages had matured (Metal)

it would be possible to create a decent cross-compiler by **adding a frontend that parses HLSL** and some code to handle unique HLSL features.

YACCGLO™ (Yet another cross-compiler based on glsl-optimizer)

- use glsl-optimizer as a basis for a cross-compiler
- DX11 HLSL frontend, Metal and GLSL backends

```
Texture2D<float4> inputData : register (t0);

float4 constant;
static float4 global = float4(0.25f, 0.5f, 0.
    75f, 1.0f);

struct Input
{
    uint vertexId : SV_VertexID;
    uint instanceId : SV_InstanceID;
    float4 pos : TEXCOORD0;
};

struct Output
{
    float4 pos : SV_Position;
    float4 data : TEXCOORD0;
    float4 data2 : TEXCOORD1;
};

float getDefaultValue(uint def=8)
{
    return def;
}

Output vsSiggraph(Input input) : SV_Position
{
    Output output = (Output)0;

    output.pos = input.pos;
    output.data = inputData[uint2(input.
        vertexId, input.instanceId)] *
        getDefaultValue() * constant *
        global;

    return output;
}
```

```
// GLSL

uniform sampler2D inputData;
uniform vec4 constant;
layout(location=0) in vec4 TEXCOORD0_in;
out vec4 TEXCOORD0_out;
out vec4 TEXCOORD1_out;
void main ()
{
  uvec2 tmpvar_1;
  tmpvar_1.x = uint(gl_VertexID);
  tmpvar_1.y = uint(gl_InstanceID);
  gl_Position = TEXCOORD0_in;
  TEXCOORD0_out = ((vec4(2.0, 4.0, 6.0, 8.0) *
      constant) * texelFetch (inputData, ivec2(
      tmpvar_1), int(0)));
  TEXCOORD1_out = vec4(0.0, 0.0, 0.0, 0.0);
}

// Metal

struct Input {
    float4 TEXCOORD0_in [[attribute(0)]];
};
struct Output {
    float4 gl_Position [[position]];
    float4 TEXCOORD0_out [[user(TEXCOORD0_out)]];
    float4 TEXCOORD1_out [[user(TEXCOORD1_out)]];
};
struct UniformBuffer {
    float4 constant;
};
vertex Output blsl_main (
    Input inputs [[stage_in]],
    constant UniformBuffer& uniforms [[buffer(0)]],
    uint gl_InstanceID [[instance_id]],
    uint gl_VertexID [[vertex_id]],
    texture2d<float, access::sample>
        texture_inputData [[texture(0)]])
{
  Output outputs;
  uint2 tmpvar_1 = uint2();
  tmpvar_1.x = uint(gl_VertexID);
  tmpvar_1.y = uint(gl_InstanceID);
  outputs.gl_Position = inputs.TEXCOORD0_in;
  outputs.TEXCOORD0_out = ((float4(2.0, 4.0, 6.0, 8.0
      ) * float4(uniforms.constant)) * float4(
      texture_inputData.read(tmpvar_1, 0)));
  outputs.TEXCOORD1_out = float4();
  return outputs;
}
```

```
{
    "reflection" : {
        "compute_info" : {
            "num_threads_x": 0,
            "num_threads_y": 0,
            "num_threads_z": 0
        },
        "textures" : [
            {"name": "inputData", "dst_index": 0, "
                src_index": 0, "src_register_type":
                "t"}
        ],
        "samplers" : [
        ],
        "buffers" : [
            {"name": "uniforms", "dst_index": 0, "
                src_index": 0, "src_register_type":
                "b"}
        ],
        "uniforms" : [
            {"name": "constant", "offset": 0, "size"
                : 16}

        ]
    },
    "metrics" : {
    "alu" : 5,
    "tex" : 1,
    "flow" : 0
    }
}
```

a DX11 style syntax compiler that supports compiling all of the shaders, including compute shaders.

The result of a compilation is a Metal or GLSL source file, with a json file containing reflections and meta info.

小结

- a bleeding edge engine can run on today's mobile hardware.
- a straight implementation of the engine, cross compiling all shaders with minor divergence.
- many tile memory specific opt can be done without diverging code from PC / consoles
- cross-compilation is key for codebase coordination and performance tuning

Approach: Get the full picture before diving into details

We prefer to disable slow features until we attack them and attack the ones giving the most bang for the buck first. We believe we're at a point now when we have the full picture and can start diving into the details.

Surprise Driven!!

Check out "(2015) #siggraph2015 Frostbite on Mobile.pptx"
for further readings on Light-tiling (Forward/Deferred) / PBR / render-pass merging

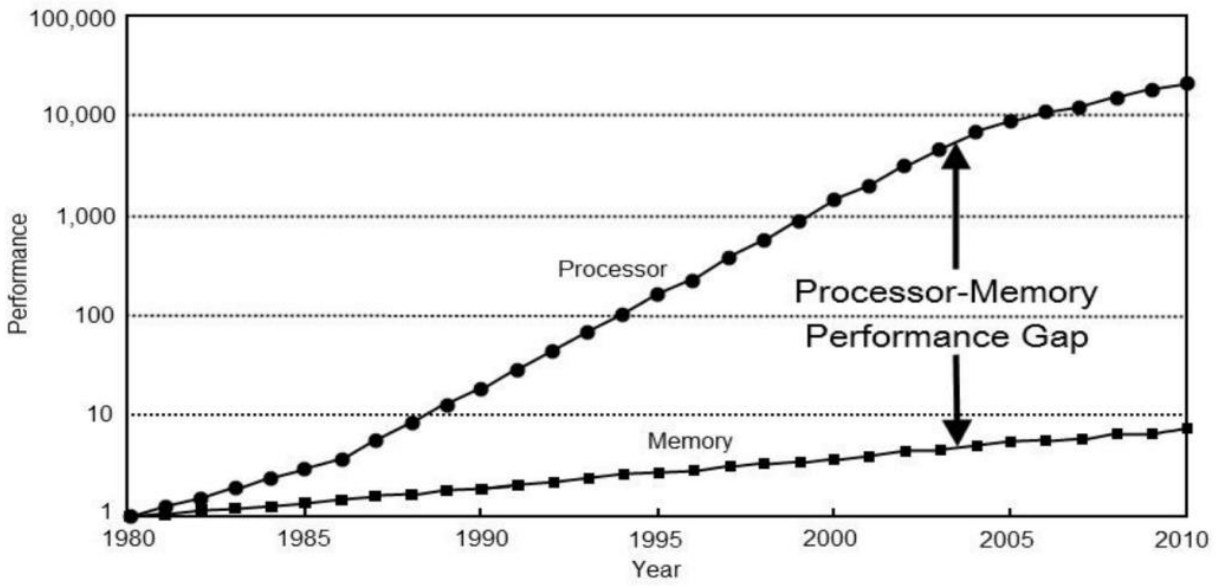
目录

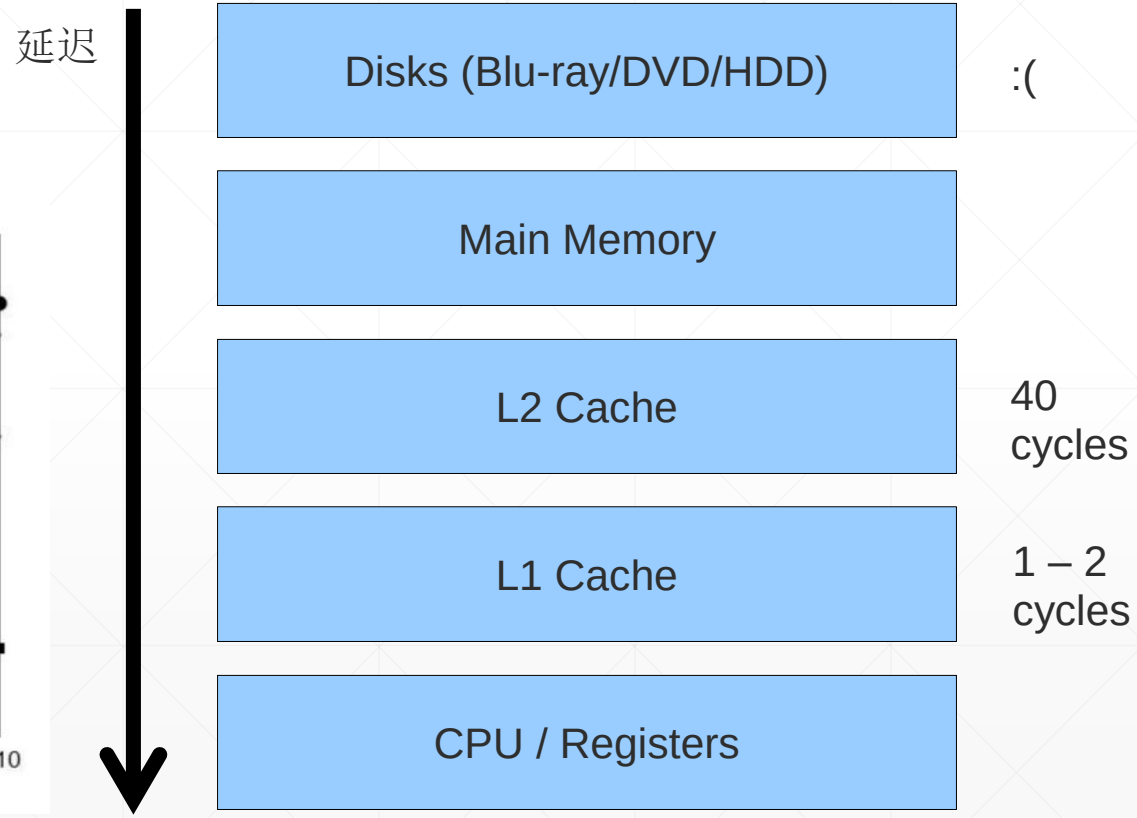
什么是“围绕数据设计”(Data-Oriented Design)?

- Object-Oriented vs Data-Oriented
- 在设计和实现一个系统时
  - 不再通过“对象”来建模和表达概念
  - 把关注点从“对象和交互”转移到“数据和读写”
- 仍然封装，只是…
  - 不是形成一套类体系
  - 而是围绕活跃的数据集构筑访问模型

为什么需要“围绕数据设计”？

Reason #1: Performance

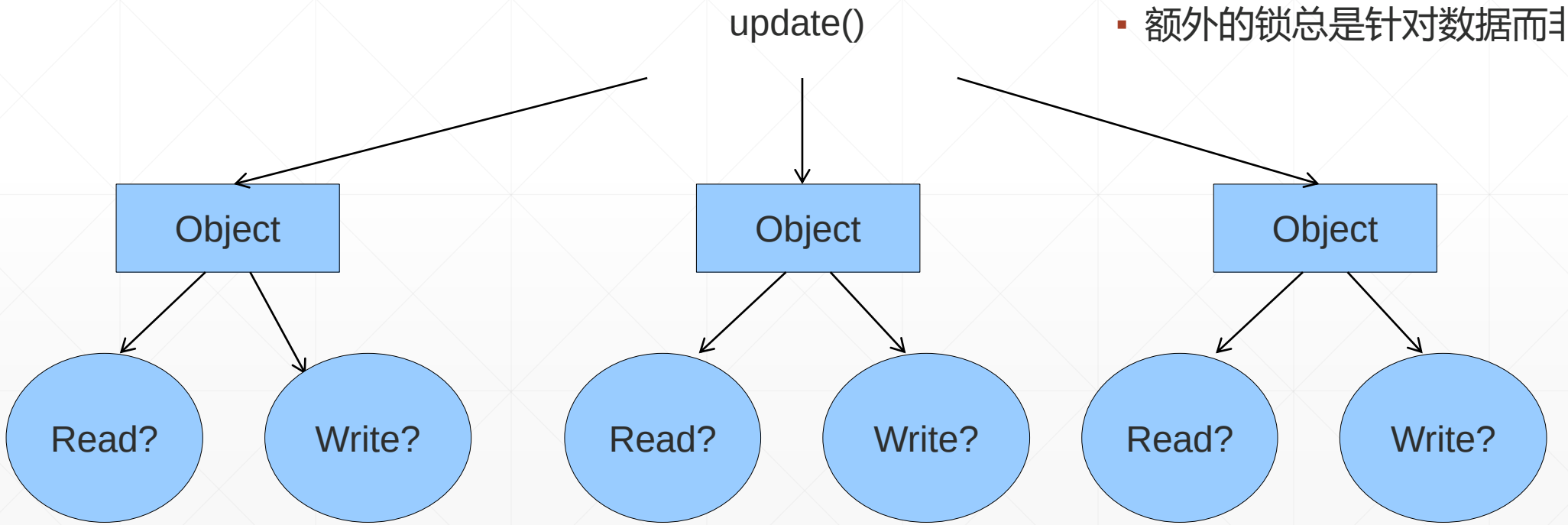


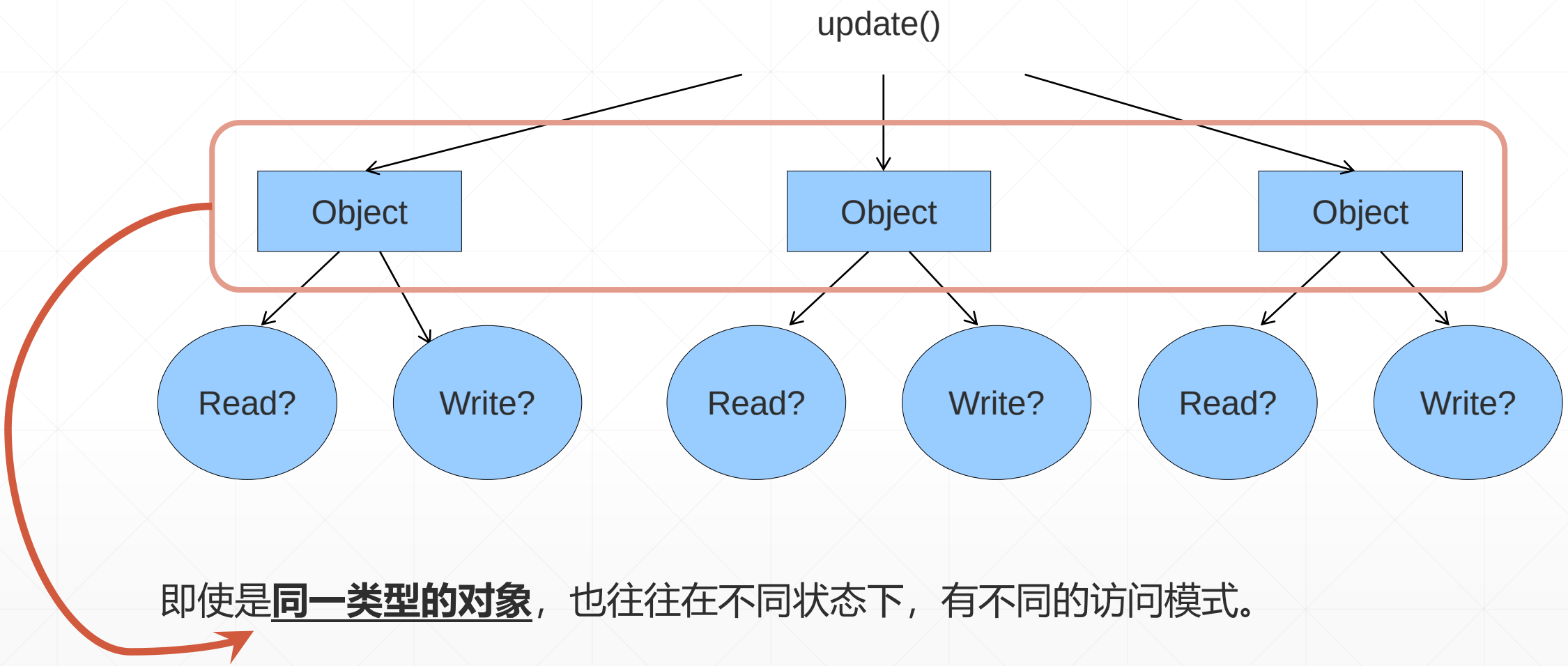


为什么需要“围绕数据设计”？

Reason #2: Multithreading

- 对现有代码做线程化时
  - 数据组织比对象模型更重要
  - 额外的锁总是针对数据而非行为





即使是**同一类型的对象**，也往往在不同状态下，有不同的访问模式。

对象这一层代码抽象失去了意义。在线程同步时，无法反映我们关心的交互和状态了。

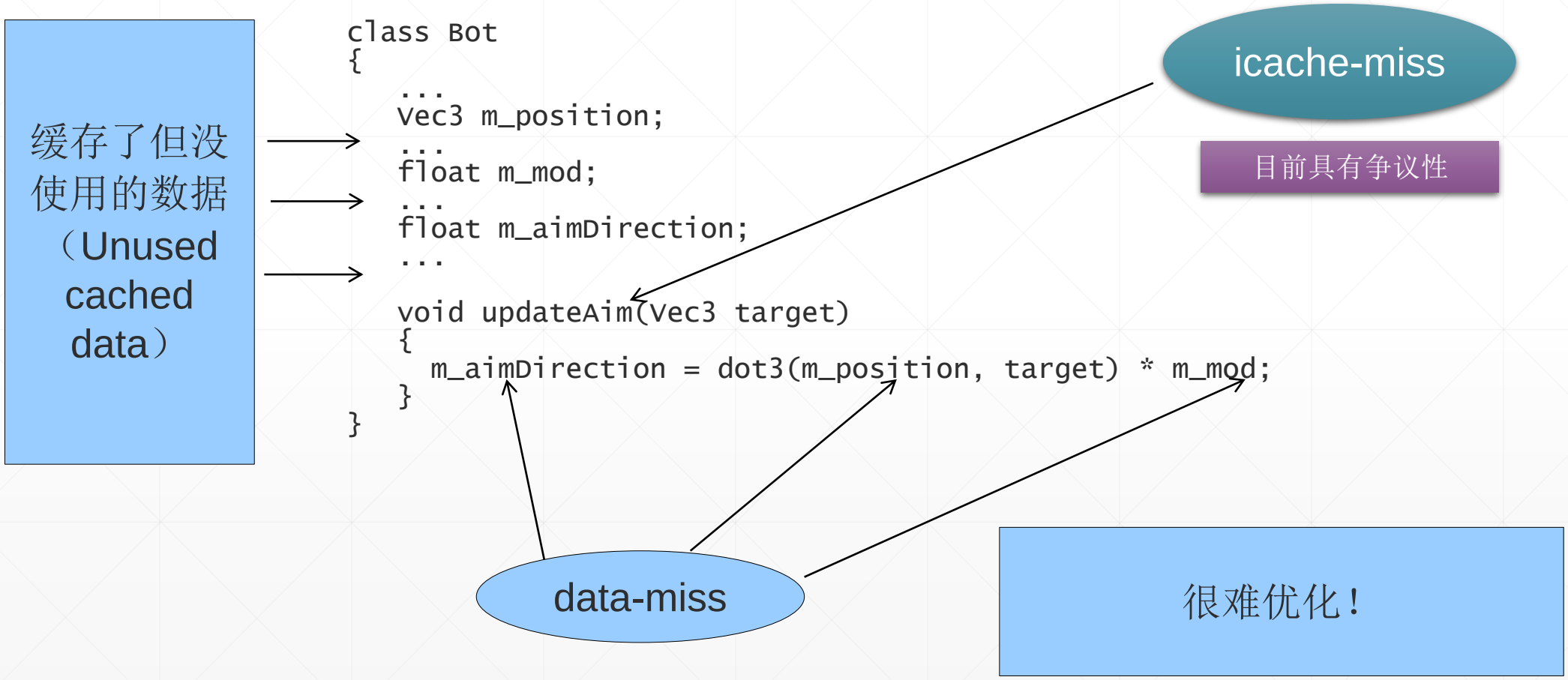
围绕数据设计的着眼点

- 现代体系下的内存访问敏感性强
  - 往往直接影响全局性能 - 提供紧凑的数据，理解和配合 Cache 的行为越来越重要。
- 当把行为线程化时
  - 对数据读写的明确约定能够极大地简化设计，避免不必要的锁。
  - 良好的面向对象设计，会把逻辑和数据耦合在一起，而线程同步往往只关心数据本身。耦合带来维护负担。（POD-struct 的同步几乎总是比对象同步要简明）

C/Go 的数据同步往往比 C++ 的冗余代码少，更容易维护

- 完整而独立的数据总是比散落在不同对象里的单个状态更容易保证一致性，更容易测试和调试

举例 - OOD

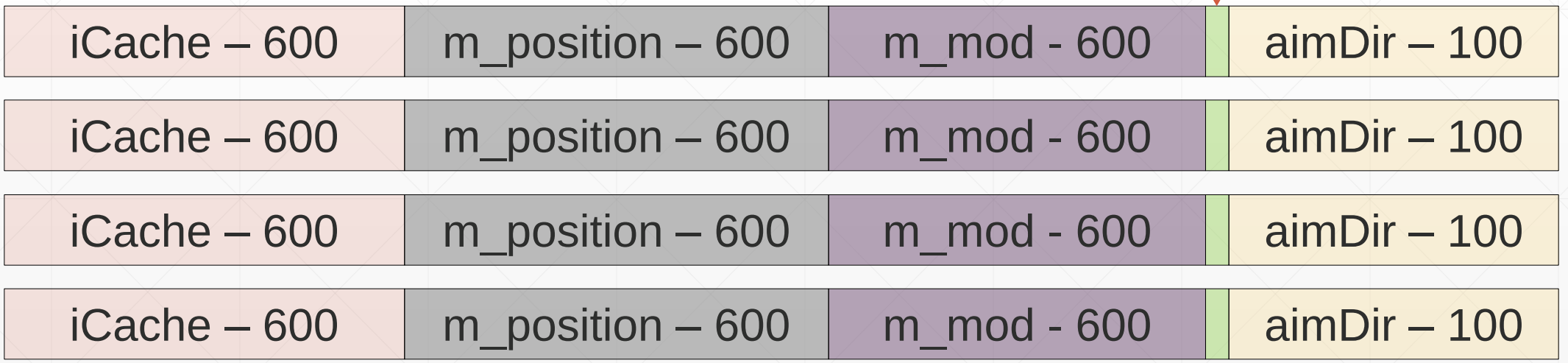

缓存了但没使用的数据（Unused cached data）
class Bot
{
...
Vec3 m_position;
...
float m_mod;
...
float m_aimDirection;
...
void updateAim(Vec3 target)
{
m_aimDirection = dot3(m_position, target) * m_mod;
}
}
icache-miss
目前具有争议性
data-miss
很难优化！

举例 - OOD

假设以下的函数调了 4 次 (4 个不同的 Bots)

```
void updateAim(Vec3 target)
{
  m_aimDirection = dot3(m_position, target) * m_mod;
}
```

~20 cycles



7680

98.96% 的时间花费在实际运算逻辑之外，惊人的浪费

举例 - OOD

- 以从后向前的方式来设计 （back-to-front）
  - 专注于围绕**输出的数据**来组织代码
- 仅仅提供得到正确结果所需的最少量的数据 (minimal amount of input)

举例 - OOD

```
void updateAims(float* aimDir,const AimingData* aim,
                Vec3 target, uint count)
{
    for (uint i = 0; i < count; ++i)
    {
        aimDir[i] = dot3(aim->positions[i],target) * aim->mod[i];
    }
}
```

变了的部分

只采集那些需要的数据	写入到线性数组里
紧凑地遍历所有的数据	实际运算代码保持不变

额外的好处！

代码独立 (standalone) 而自洽 (self-contain)

自动地与对象解耦合 (like <algorithm>)

举例 - OOD

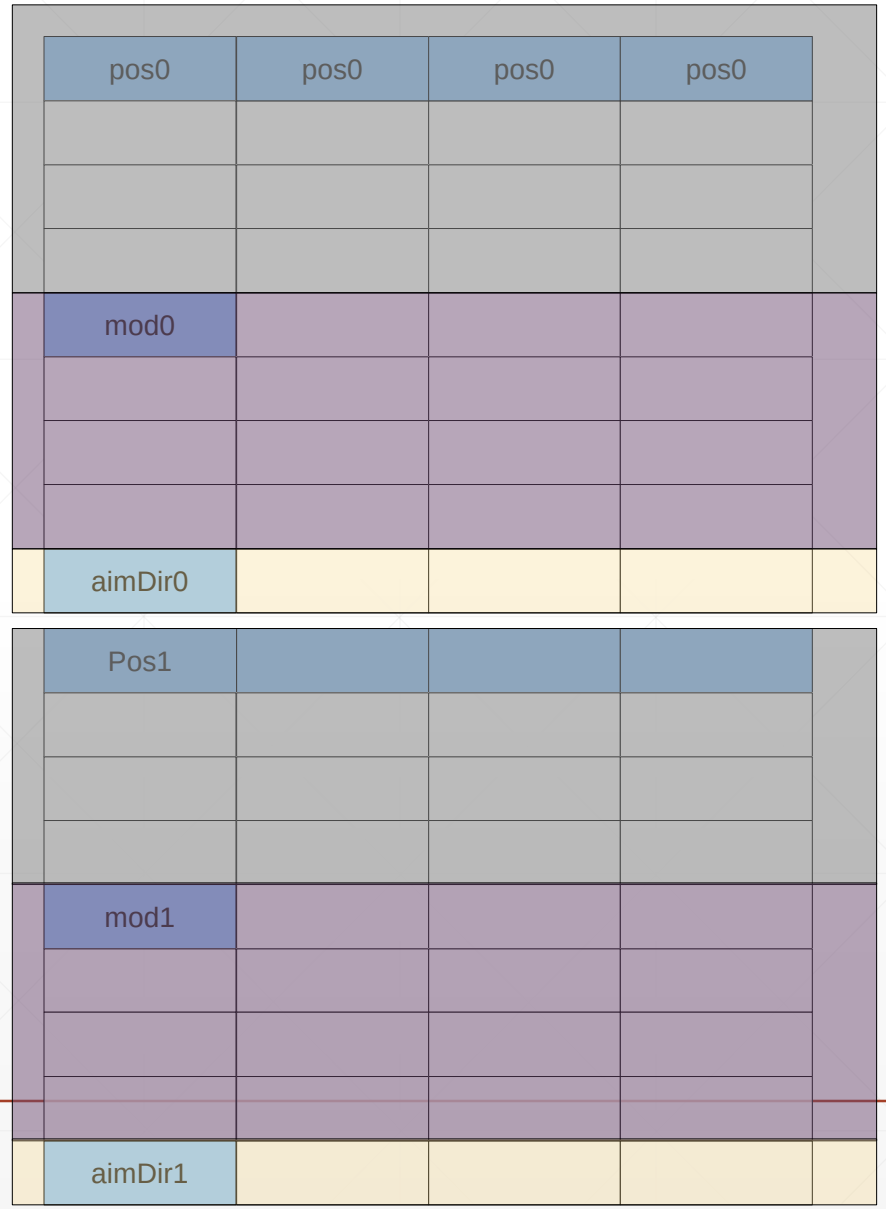
```
void updateAims(float* aimDir, const AimingData* aim,
                Vec3 target, uint count)
{
  for (uint i = 0; i < count; ++i)
  {
    aimDir[i] = dot3(aim->positions[i], target) * aim->mod[i];
  }
}
```

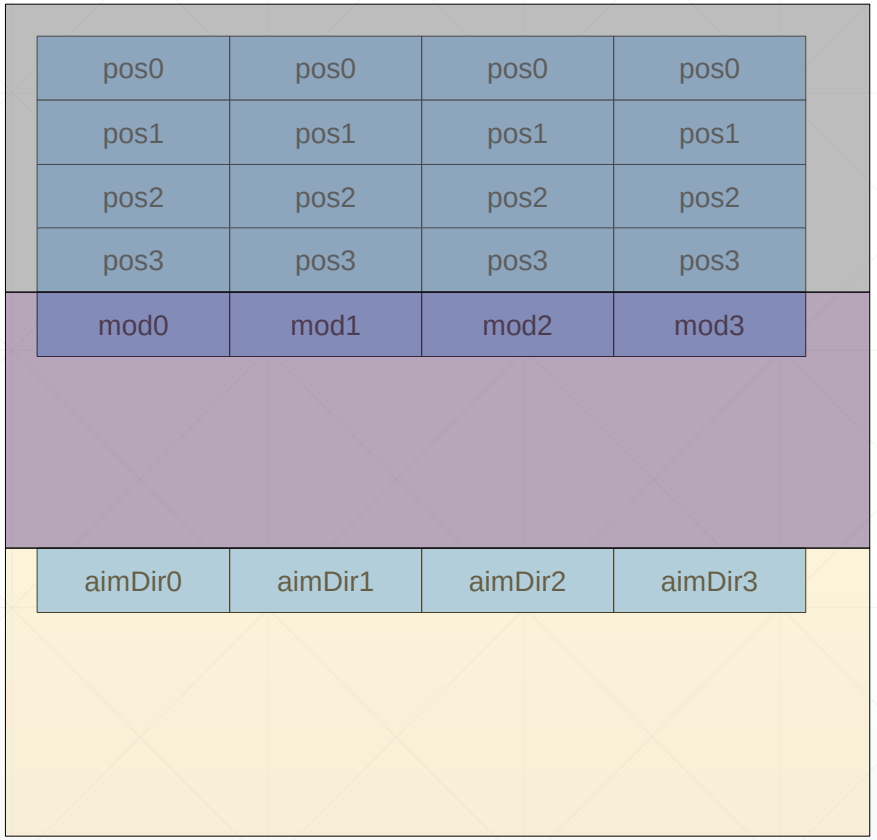
~20 cycles

1980

消耗在内存寻址上的时间降低到原来的 25% (1900 ~ 7600)

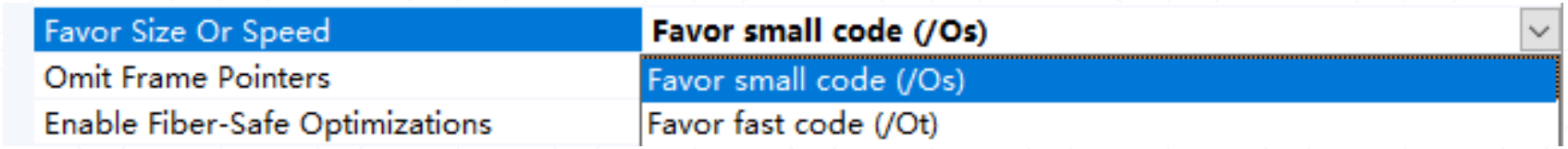
内存布局对比





每块是 128 字节的 cache line

核心关注点是内存的利用效率

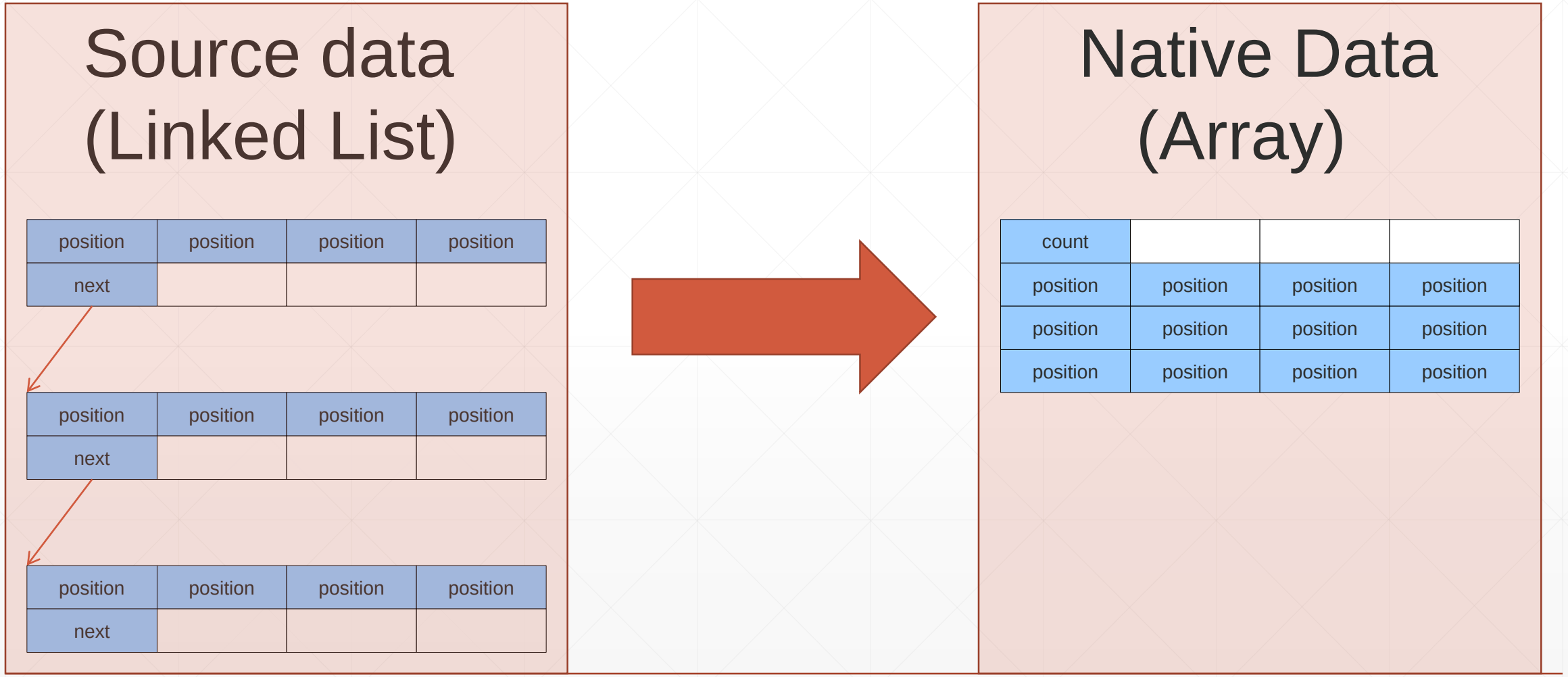
- 优先针对数据优化，而不是代码
  - 

- 绝大部分代码的性能取决于内存访问 (**bound by memory access**)
- 用于运算的数据集 (native data) 可以跟源数据集 (source data) 分离
  - (就好像 native code 和 source code 那样)
  - 可以理解为，为了更高效的访问，我们把数据“编译”为更紧凑的组织形式

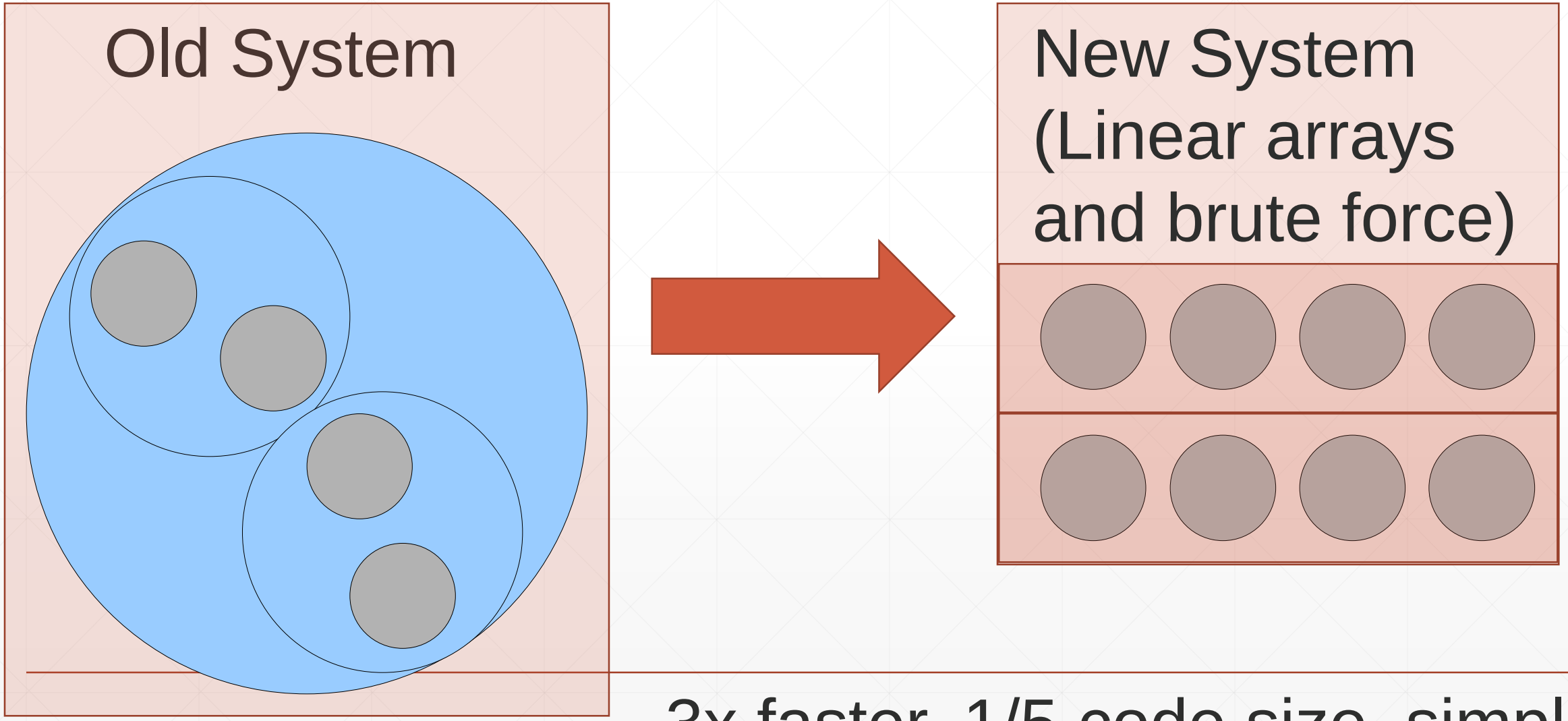
Not everything needs to be an object.

We are doing games, we (need to) know our data.

举例 – 场景内的区域触发器



举例 – 裁剪系统

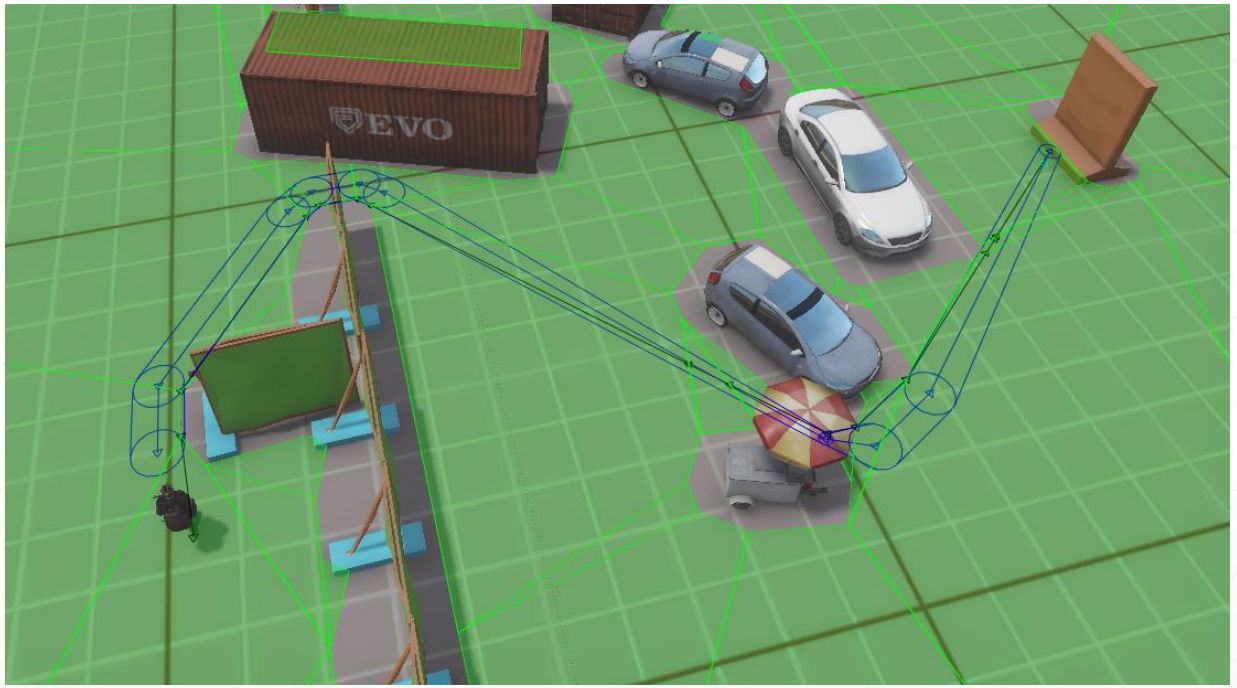


3x faster, 1/5 code size, simpler

围绕数据设计 – 收益小结

- 更好的性能
- 更简单的代码
- 让并行化更容易

交叉引用 – DOD 在AI寻路和动画上的运用



- Let's not abandon OO nor rewrite the world
- Start small, batch a bit, resolve inputs, avoid deep dives, grow from there
- Much easer to rewrite a system in a DO fashion afterwards

BE NICE TO YOUR MEMORY ☺

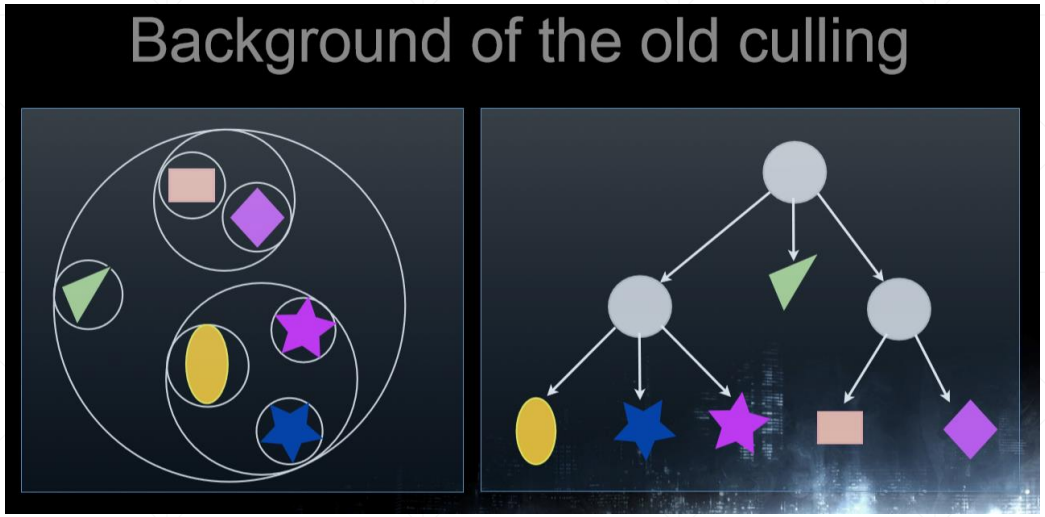
new / push_back() / insert() / resize()

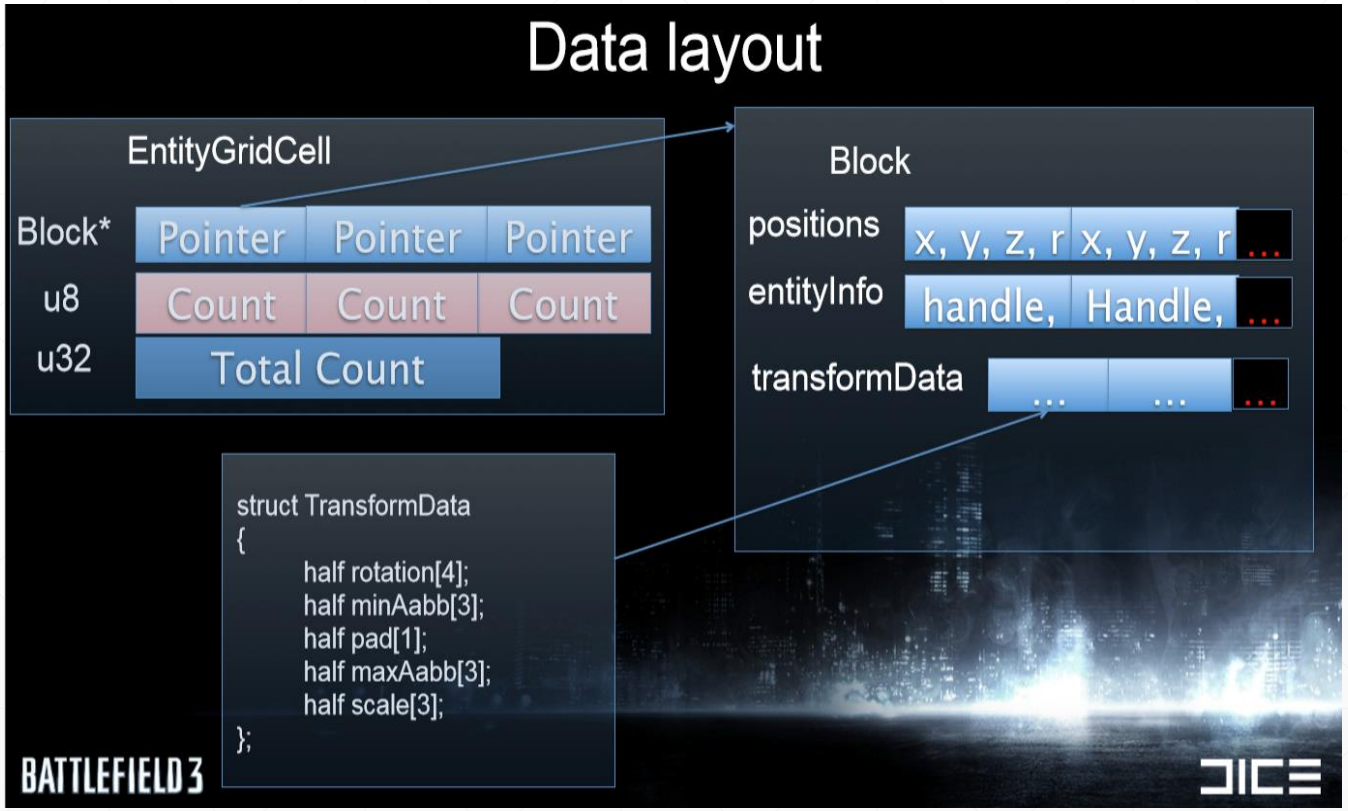
Stop and think!

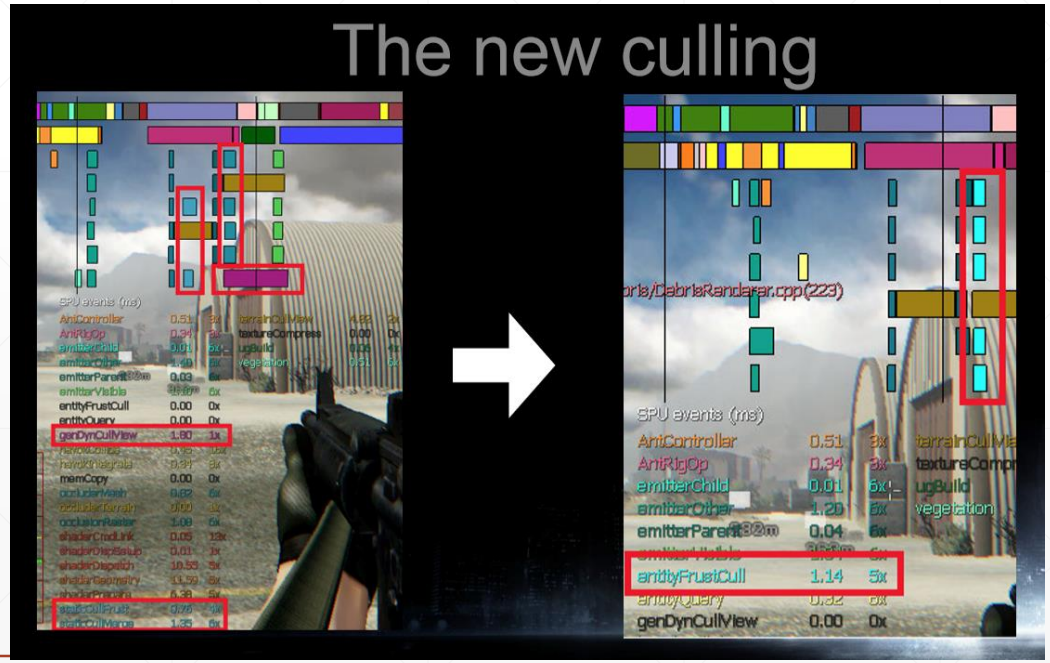
- Where is the memory allocated?
- Pre-allocated containers?
- Scratch pad?
- Can I resize()/reserve() immediately?
- Can I use Optional<T> instead of ScopedPtr<T>?
- Can I use vectors instead of list / set / map?

(2010) DICE Coders Day 2010 - AStepTowardsDataOrientation.ppt 6,384 KB

交叉引用 – DOD 在裁剪中的运用



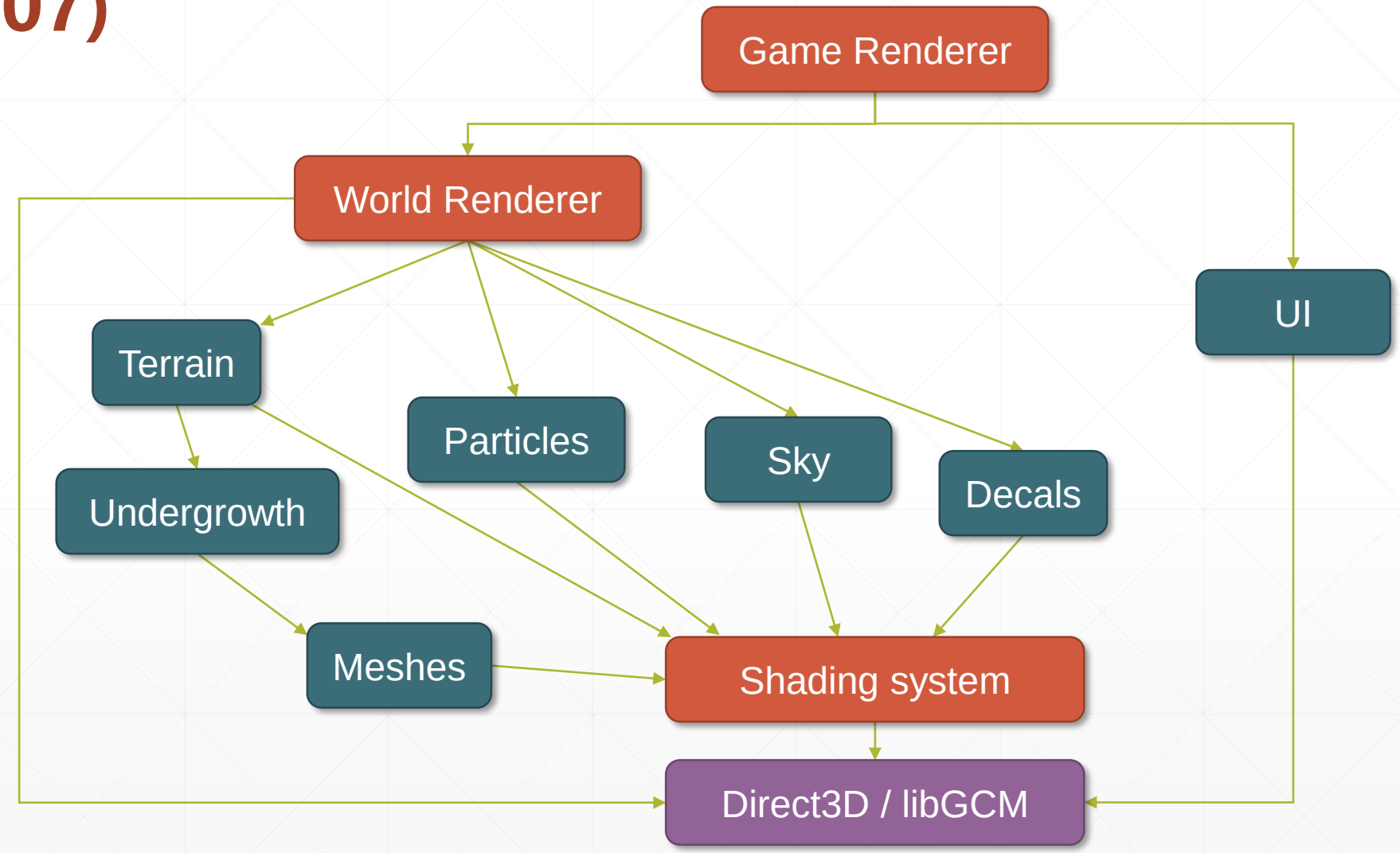




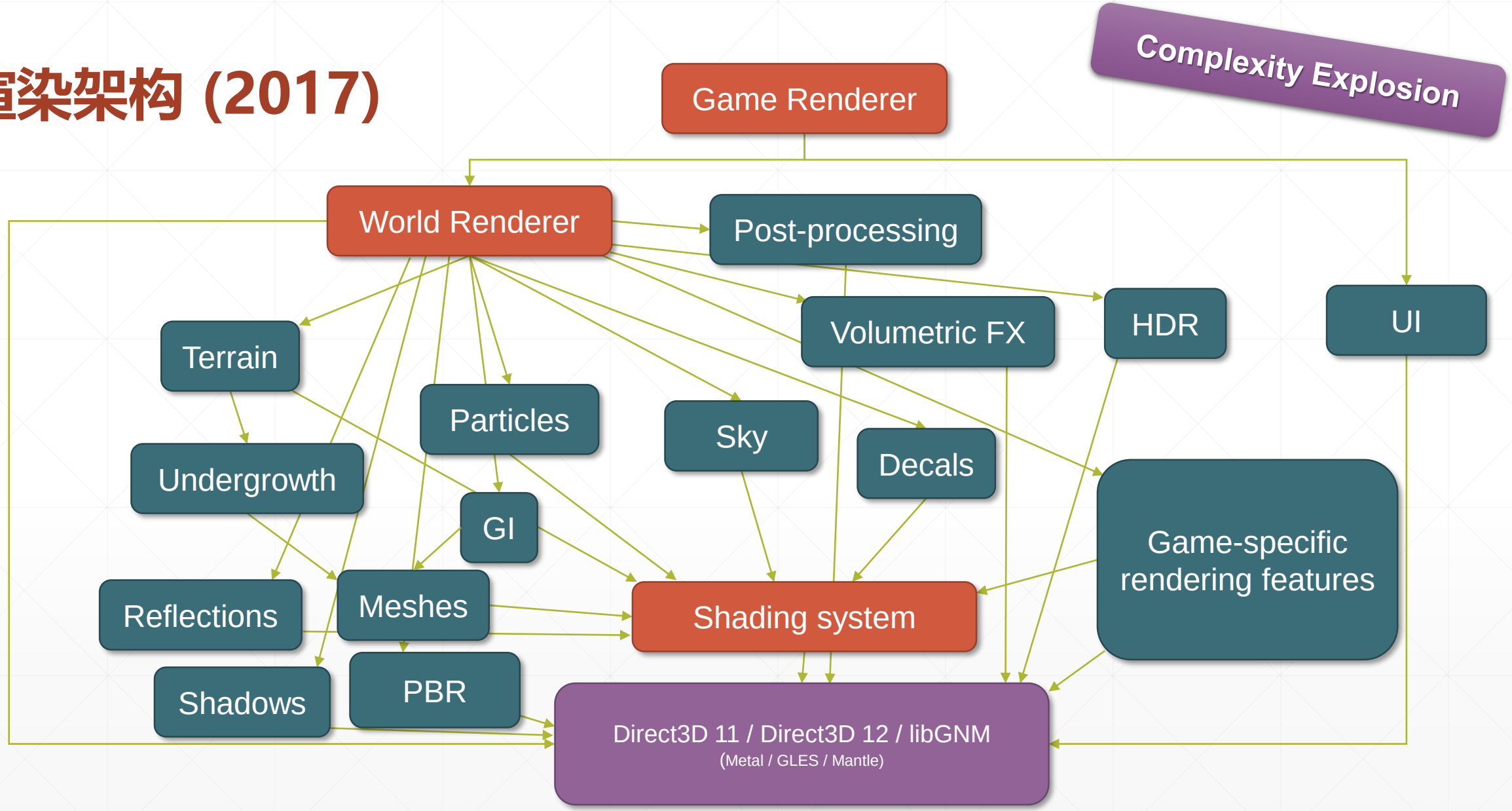
(2011) #GDC2011 Battlefield 3 & Frostbite 2 - Culling The Battlefield.pdf 6,741 KB

目录

渲染架构 (2007)


Game Renderer
World Renderer
UI
Terrain
Particles
Sky
Decals
Undergrowth
Meshes
Shading system
Direct3D / libGCM

渲染架构 (2017)


Complexity Explosion
Game Renderer
World Renderer
Post-processing
Volumetric FX
HDR
UI
Terrain
Particles
Sky
Decals
Undergrowth
GI
Game-specific rendering features
Reflections
Meshes
Shading system
Shadows
PBR
Direct3D 11 / Direct3D 12 / libGNM
(Metal / GLES / Mantle)

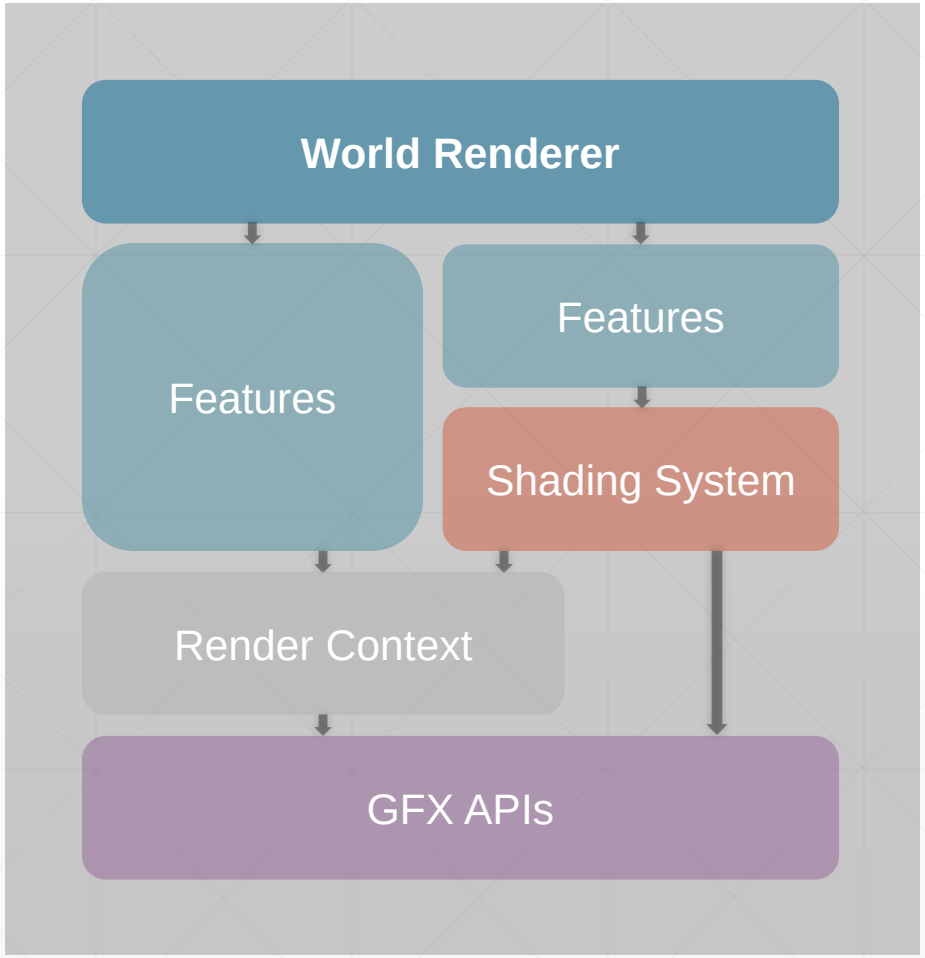
Battlefield 4 rendering passes

Aggressively Evolved Featureset

- reflectionCapture
- planarReflections
- dynamicEnvmap
- mainZPass
- mainGBuffer
- mainGBufferSimple
- mainGBufferDecal
- decalVolumes
- mainGBufferFixup
- msaaZDown
- msaaClassify
- lensFlareOcclusionQueries
- lightPassBegin
- cascadedShadowmaps
- spotlightShadowmaps
- downsampleZ
- linearizeZ
- ssao
- hbaoHalfZ
- hbao
- ssr
- halfResZPass
- halfResTransp
- mainDistort
- lightPassEnd
- mainOpaque
- linearizeZ
- mainOpaqueEmissive
- mainTransDecal
- fgOpaqueEmissive
- subsurfaceScattering
- skyAndFog
- hairCoverage
- mainTransDepth
- linerarizeZ
- mainTransparent
- halfResUpsample
- motionBlurDerive
- motionBlurVelocity
- motionBlurFilter
- filmicEffectsEdge
- spriteDof
- fgTransparent
- lensScope
- filmicEffects
- bloom
- luminanceAvg
- finalPost
- overlay
- fxaa
- smaa
- resample
- screenEffect
- hmdDistortion

WorldRenderer challenges

- Explicit immediate mode rendering
- Explicit resource management
  - Bespoke, artisanal hand-crafted ESRAM management
  - Multiple implementations by different game teams
- Tight coupling between rendering systems
- Limited extensibility
- Game teams must fork / diverge to customize
- Organically grew from 4k to 15k SLOC
  - Single functions with over 2k SLOC
  - Expensive to maintain, extend and merge/integrate

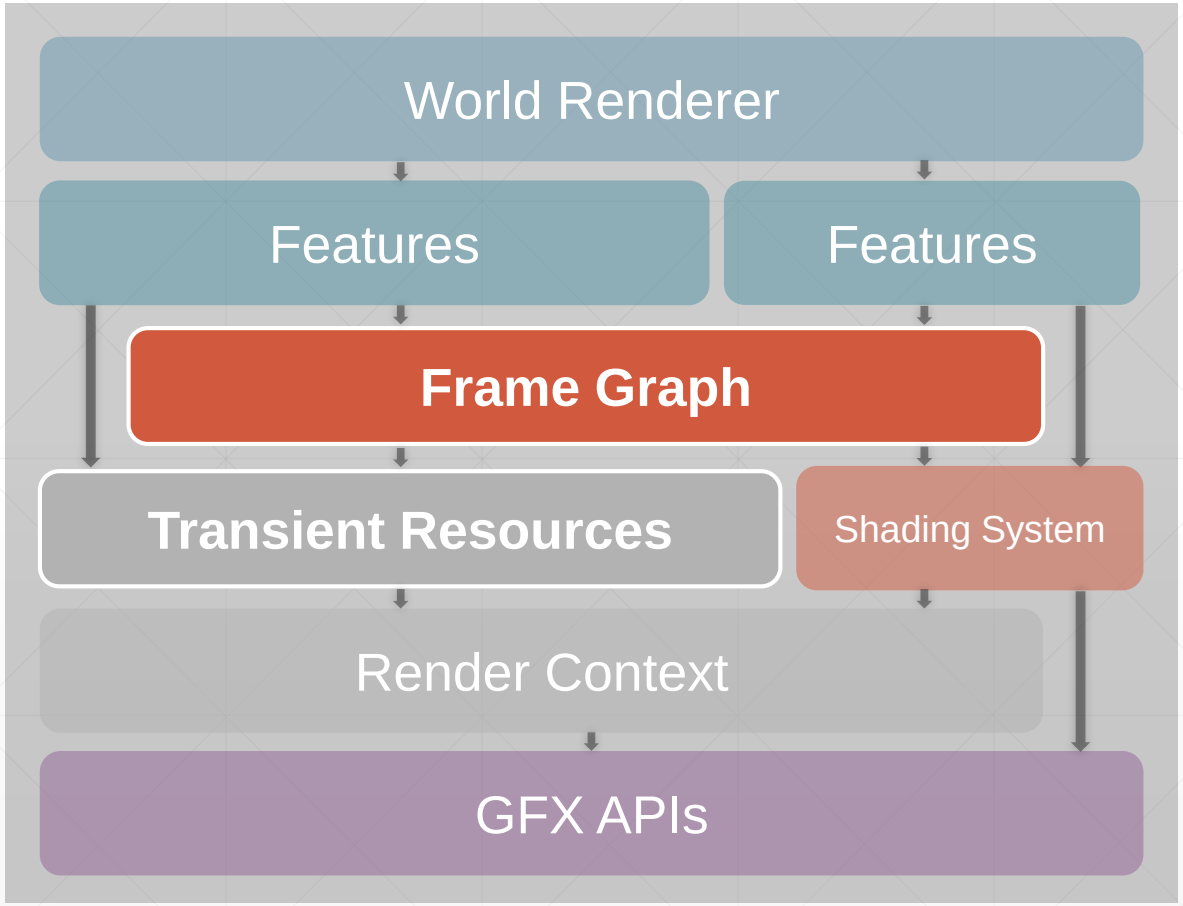


Goals

- High-level knowledge of the full frame
- Improved extensibility
  - Decoupled and composable code modules
  - Automatic resource management
- Better visualizations and diagnostics

New architectural components

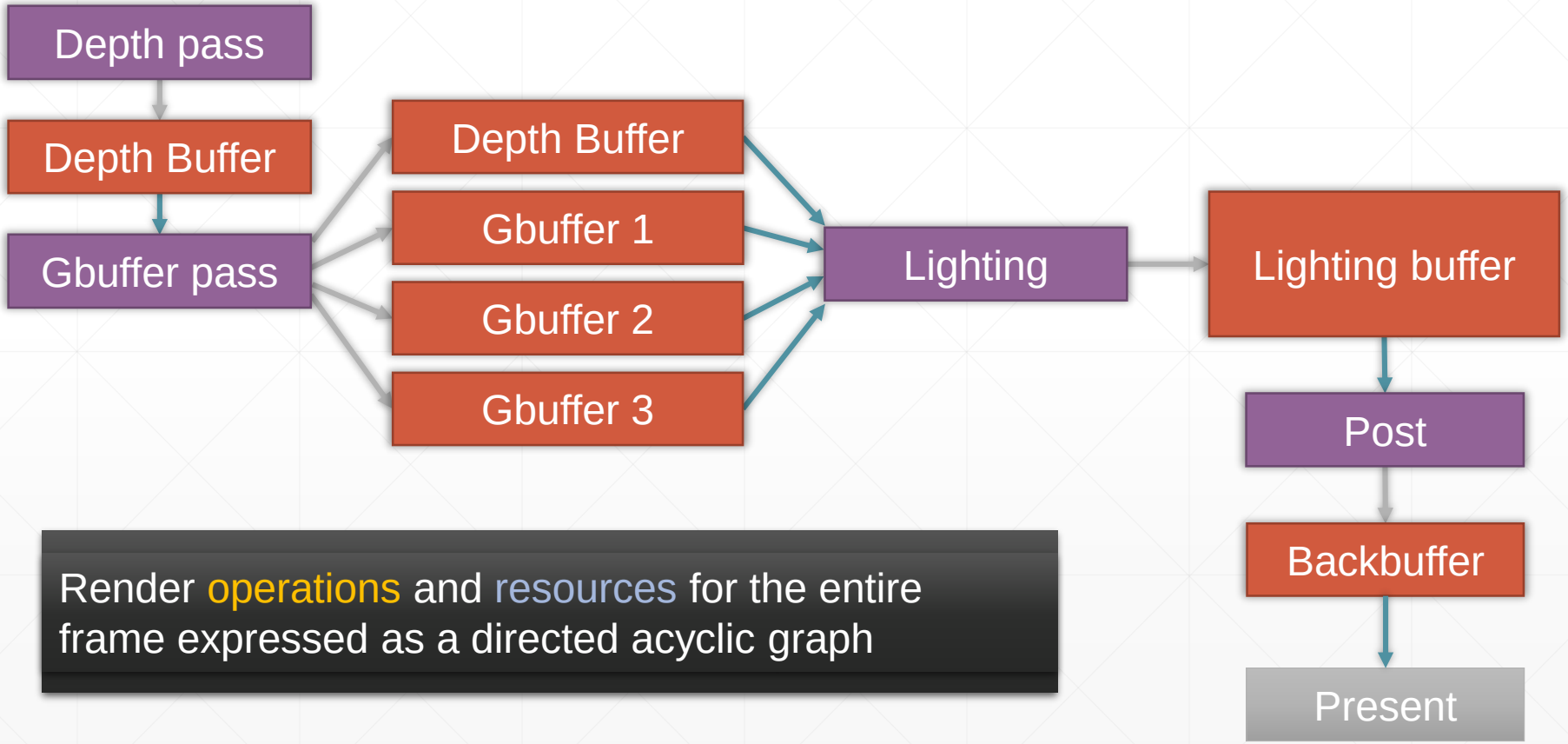
- **Frame Graph**
  - High-level representation of render passes and resources
  - Full knowledge of the frame
- **Transient Resource System**
  - Resource allocation
  - Memory aliasing



Frame Graph goals

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

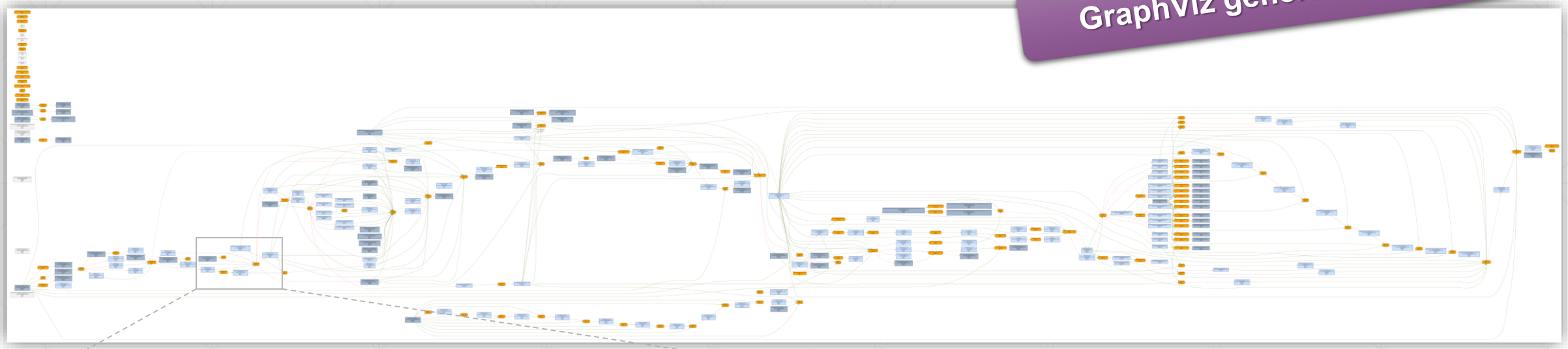
Frame Graph example

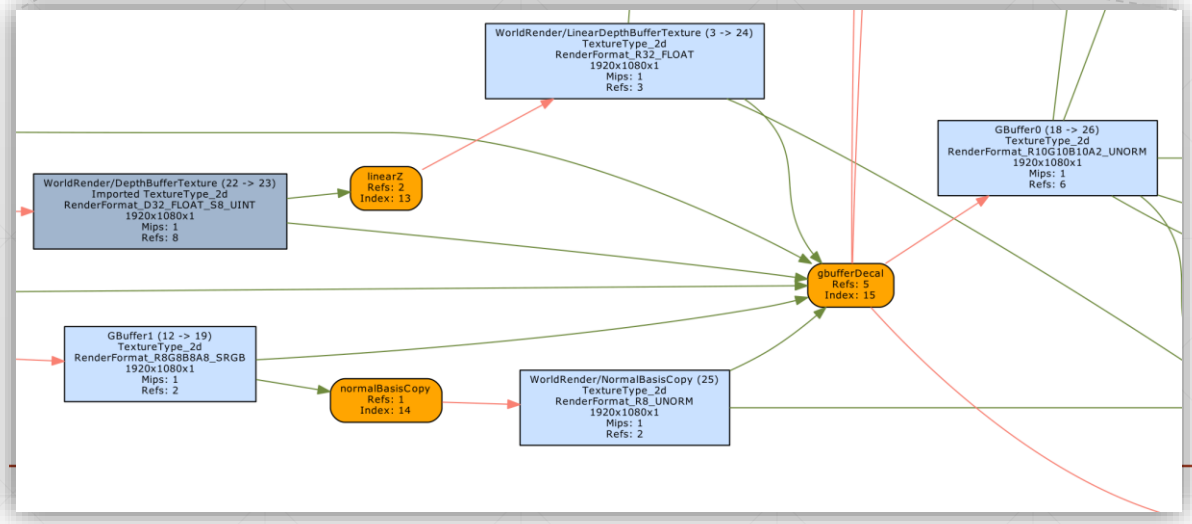


Render operations and resources for the entire frame expressed as a directed acyclic graph

Graph of a Battlefield 4 frame

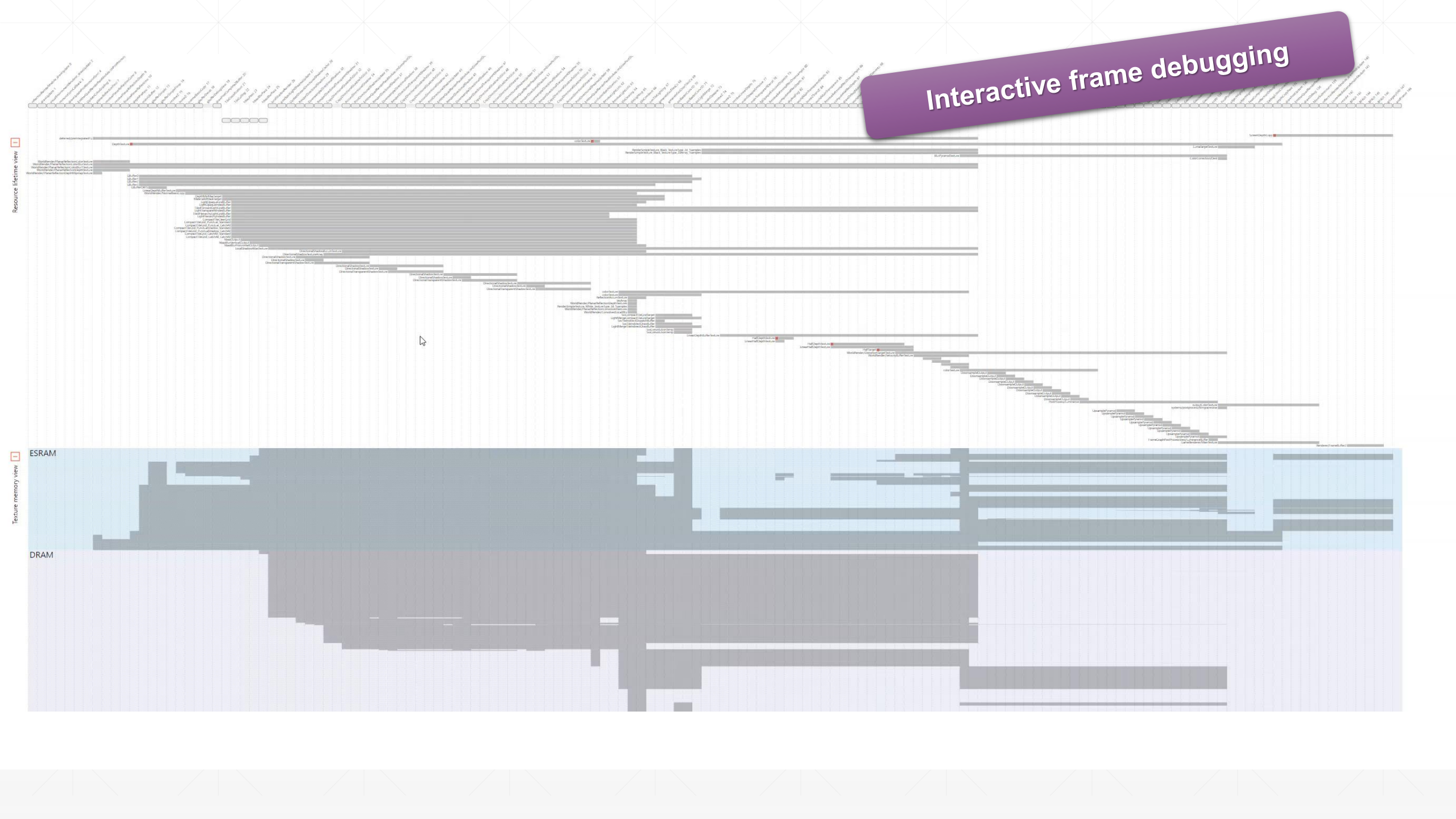
GraphViz generated on-the-fly





Typically see few hundred passes and resources

Interactive frame debugging



Frame Graph design

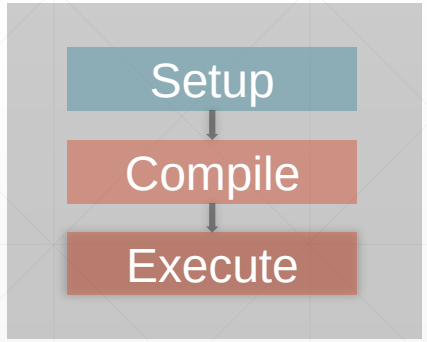
- Moving away from immediate mode rendering
- Rendering code split into passes
- Multi-phase retained mode rendering API
  1. Setup phase
  2. Compile phase
  3. Execute phase
- Built from scratch every frame
- Code-driven architecture

Frame Graph setup phase

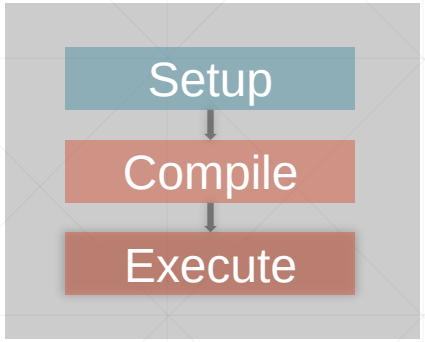
- Define render / compute passes
- Define inputs and output resources for each pass
- Code flow is similar to immediate mode rendering

not generating any GPU commands during this phase

all resources are **virtual** during graph building



Frame Graph resources



- Render passes must declare all used resources
  - Read
  - Write
  - Create
- External permanent resources are imported to Frame Graph
  - History buffer for TAA
  - Backbuffer
  - etc.

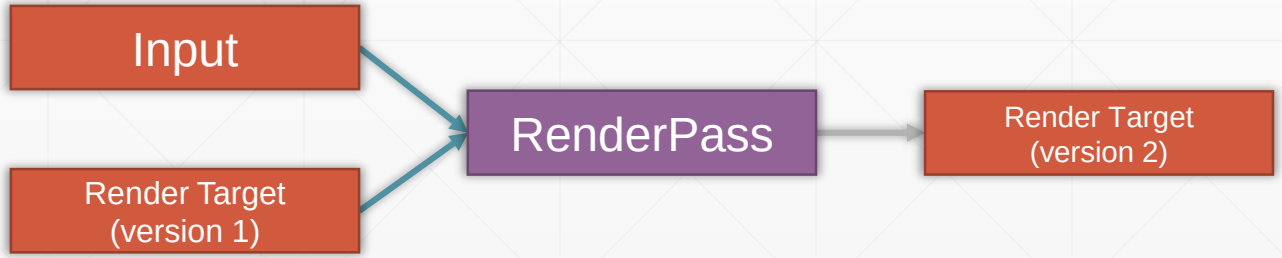
Explicitly Declared

Frame Graph resource example

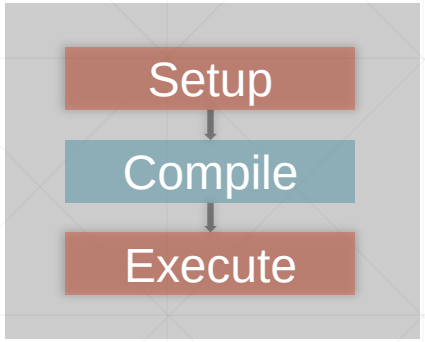
```
RenderPass::RenderPass(FrameGraphBuilder& builder)
{
        // Declare new transient resource
        FrameGraphTextureDesc desc;
        desc.width = 1280;
        desc.height = 720;
        desc.format = RenderFormat_D32_FLOAT;
        desc.initialSate = FrameGraphTextureDesc::Clear;
        m_renderTarget = builder.createTexture(desc);
}
```

Frame Graph setup example

```
RenderPass::RenderPass(FrameGraphBuilder& builder,
        FrameGraphResource input,
        FrameGraphMutableResource renderTarget)
{
        // Declare resource dependencies
        m_input = builder.read(input, readFlags);
        m_renderTarget = builder.write(renderTarget, writeFlags);
}
```

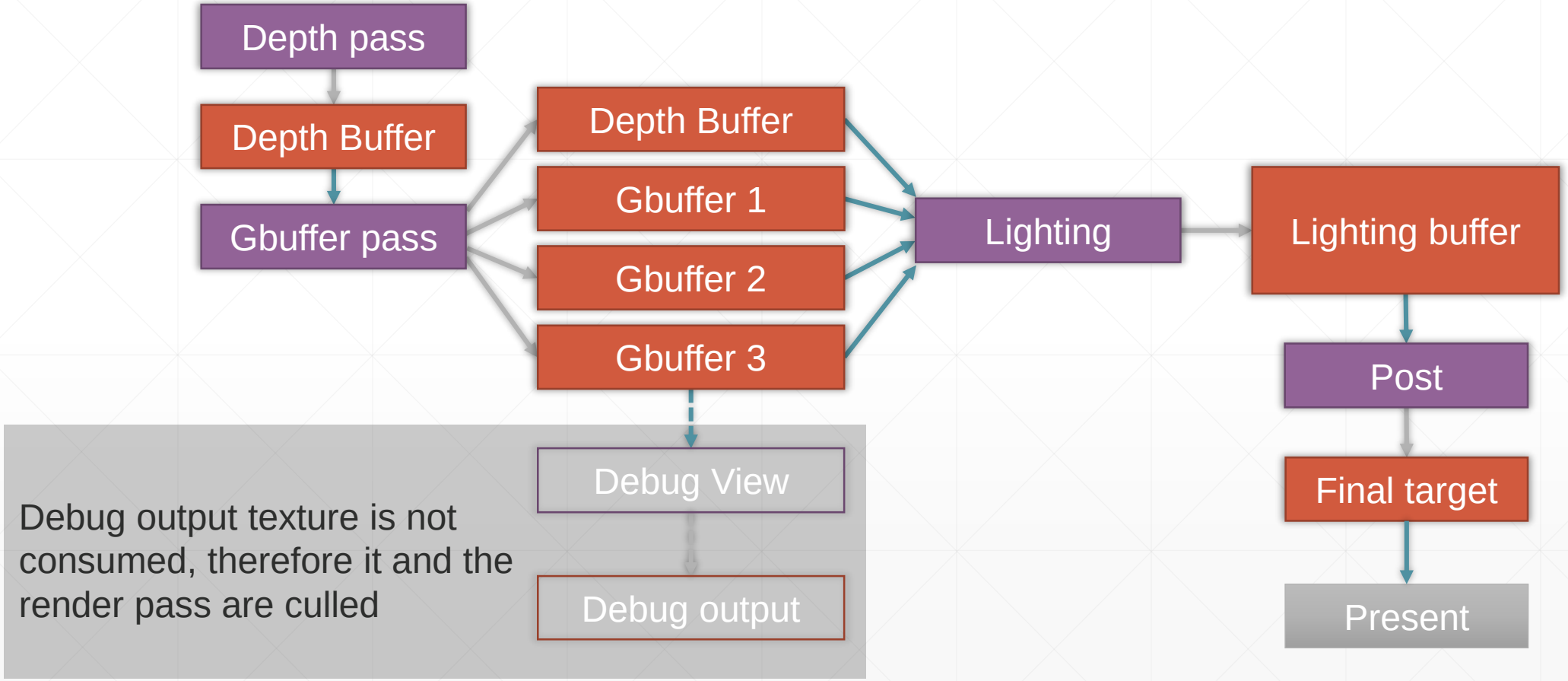


Frame Graph compilation phase



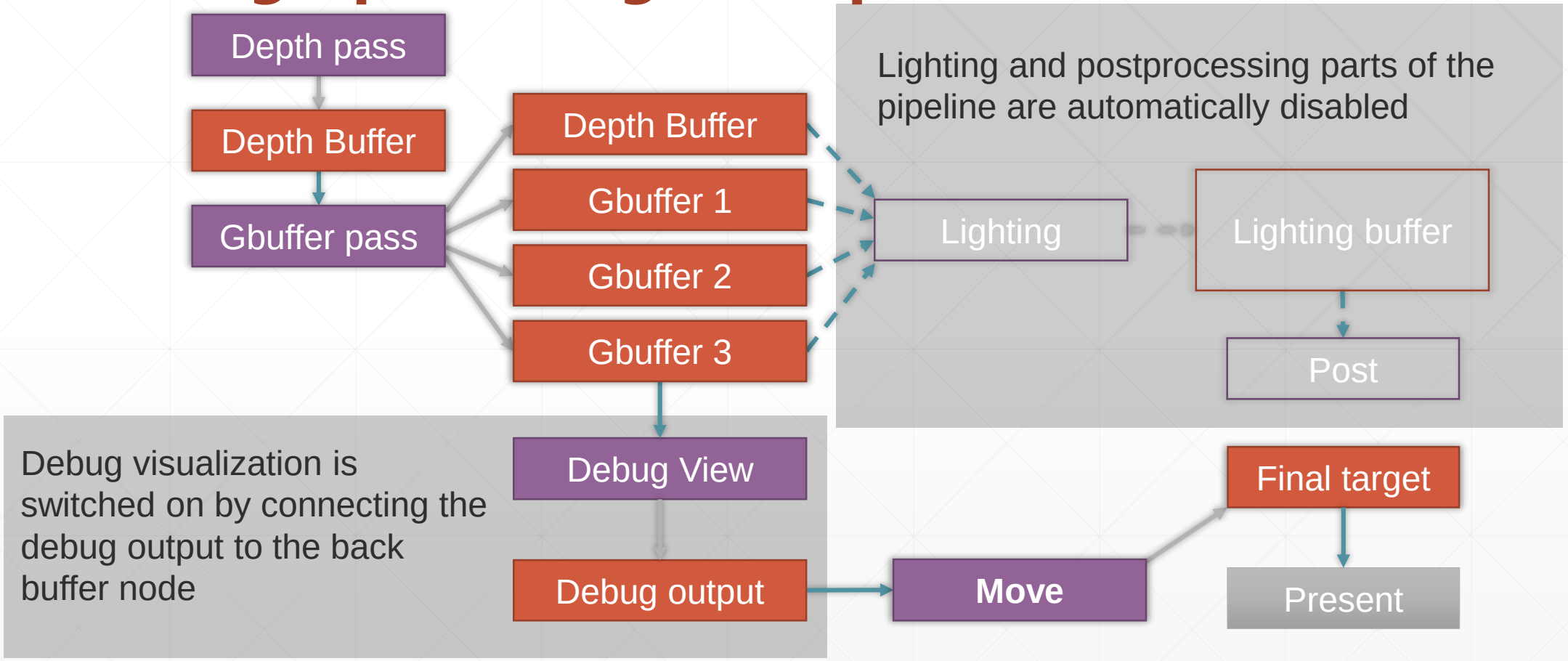
- Cull unreferenced resources and passes
  - Can be a bit more sloppy during declaration phase
  - Aim to reduce configuration complexity
  - Simplifies conditional passes, debug rendering, etc.
- Calculate resource lifetimes
- Allocate concrete GPU resources based on usage
  - Simple greedy allocation algorithm
  - Acquire right before first use, release after last use
  - Extend lifetimes for async compute
  - Derive resource bind flags based on usage

Sub-graph culling example

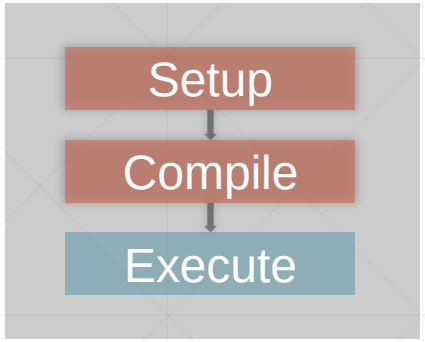


Culling is done automatically without a hand-written ‘if’

Sub-graph culling example



Frame Graph execution phase



- Execute callback functions for each render pass
- Immediate mode rendering code
  - Using familiar RenderContext API
  - Set state, resources, shaders
  - Draw, Dispatch
- Get real GPU resources from handles generated in setup phase

Pass declaration with C++

- Could just make a C++ class per RenderPass
  - Breaks code flow
  - Requires plenty of boilerplate
  - Expensive to port existing code
- Settled on C++ lambdas
  - Preserves code flow!
  - Minimal changes to legacy code
    - Wrap legacy code in a lambda
    - Add a resource usage declarations

Pass declaration with C++ lambdas

Resources

Setup

Execute
(deferred)

```
FrameGraphResource addMyPass(FrameGraph& frameGraph,
            FrameGraphResource input, FrameGraphMutableResource output)
{
            struct PassData
            {
                        FrameGraphResource input;
                        FrameGraphMutableResource output;
            };

            auto& renderPass = frameGraph.addCallbackPass<PassData>(“MyRenderPass",
            [&](RenderPassBuilder& builder, PassData& data)
            {
                        // Declare all resource accesses during setup phase
                        data.input = builder.read(input);
                        data.output = builder.useRenderTarget(output).targetTextures[0];
            },
            [=](const PassData& data, const RenderPassResources& resources, IRenderContext*
renderContext)
            {
                        // Render stuff during execution phase
                        drawTexture2d(renderContext, resources.getTexture(data.input));
            });

            return renderPass.output;
}
```

Render modules

- Two types of render modules:
  1. Free-standing stateless functions
     - Inputs and outputs are Frame Graph resource handles
     - May create nested render passes
     - Most common module type in Frostbite
  2. Persistent render modules
     - May have some persistent resources (LUTs, history buffers, etc.)
- WorldRenderer still orchestrates high-level rendering
  - Does not allocate any GPU resources
  - Just kicks off rendering modules at the high level
  - Much easier to extend
  - Code size reduced from 15K to 5K SLOC

Communication between modules

- Modules may communicate through a blackboard
  - Hash table of components
  - Accessed via component Type ID
  - Allows controlled coupling

```
void BlurModule::renderBlurPyramid(
                FrameGraph& frameGraph,
                FrameGraphBlackboard& blackboard)
{
                // Produce blur pyramid in the blur module
                auto& blurData = blackboard.add<BlurPyramidData>();
                addBlurPyramidPass(frameGraph, blurData);
}
```

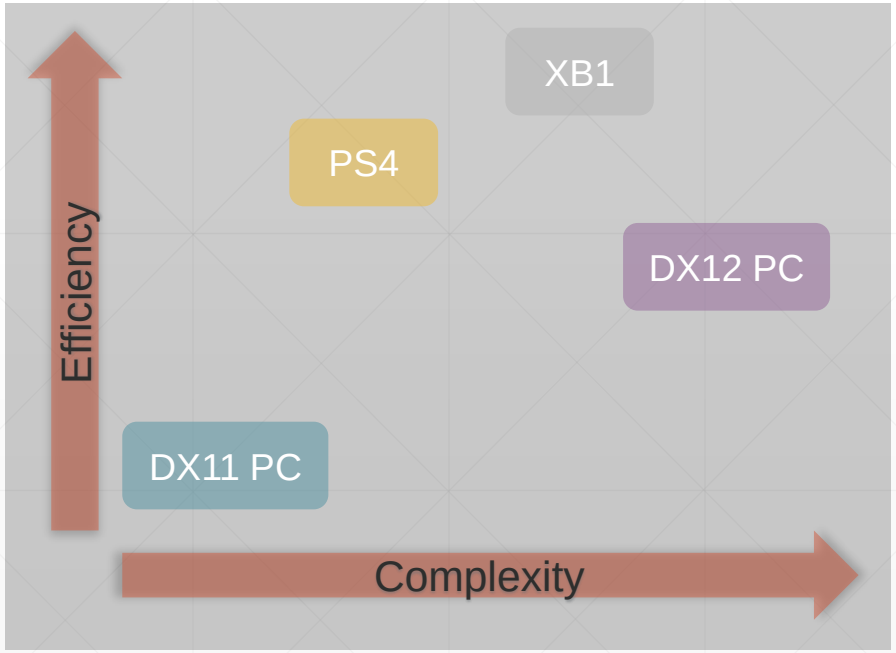
```
#include ”BlurModule.h”
void TonemapModule::createBlurPyramid(
                FrameGraph& frameGraph,
                const FrameGraphBlackboard& blackboard)
{
                // Consume blur pyramid in a different module
                const auto& blurData = blackboard.get<BlurPyramidData>();
                addTonemapPass(frameGraph, blurData);
}
```

Transient resource system

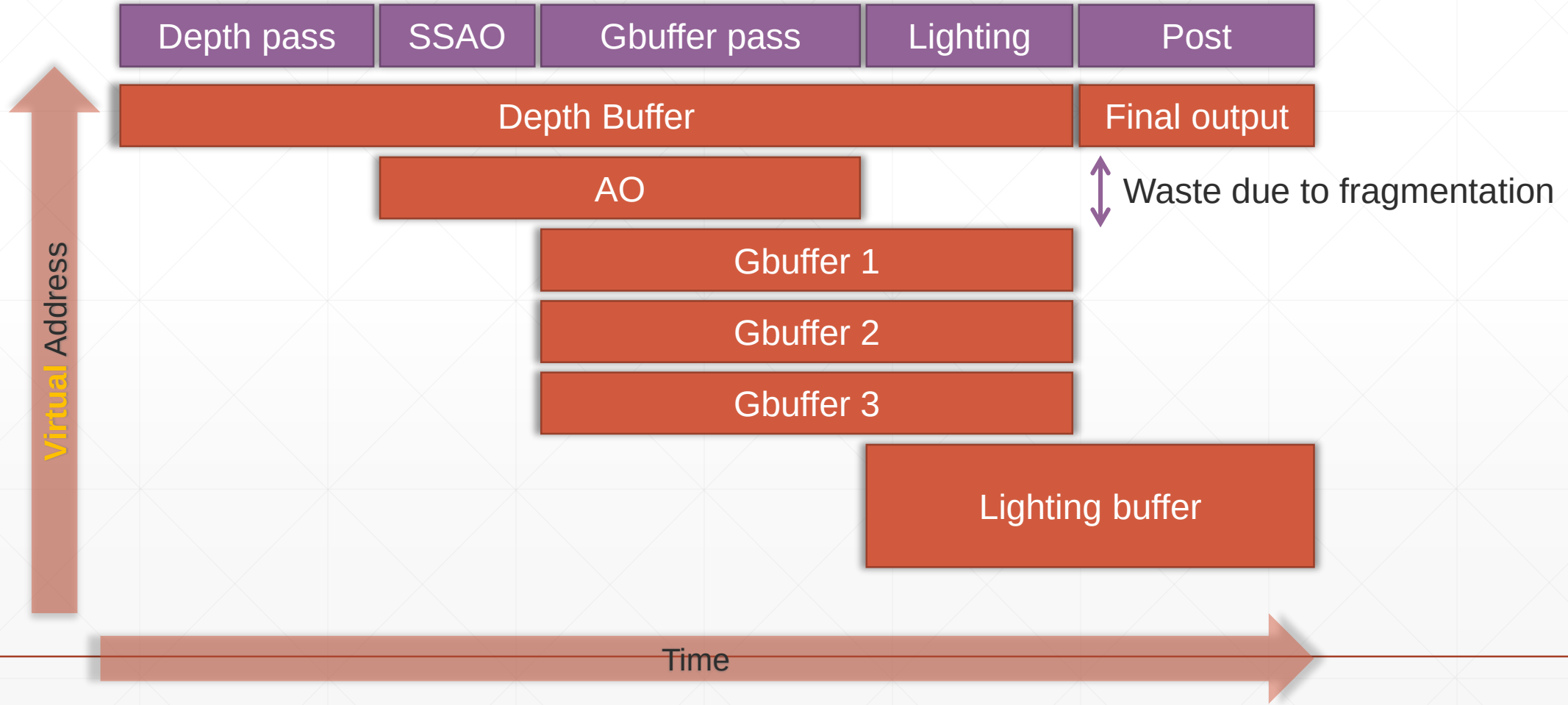
- **Transient** /ˈtranzɪənt/ *adjective*
  Lasting only for a short time; impermanent.
- Resources that are alive for no longer than one frame
  - Buffers, depth and color targets, UAVs
  - Strive to minimize resource life times within a frame
- Allocate resources where they are used
  - Directly in *leaf* rendering systems
  - Deallocate as soon as possible
  - Make it easier to write self-contained features
- **Critical component of Frame Graph**

Transient resource system back-end

- Implementation depends on platform capabilities
  - Aliasing in physical memory (XB1)
  - Aliasing in virtual memory (DX12 PS4)
  - Object pools (DX11)
- Atomic linear allocator for buffers
  - No aliasing, just blast through memory
  - Mostly used for sending data to GPU
- Memory pools for textures

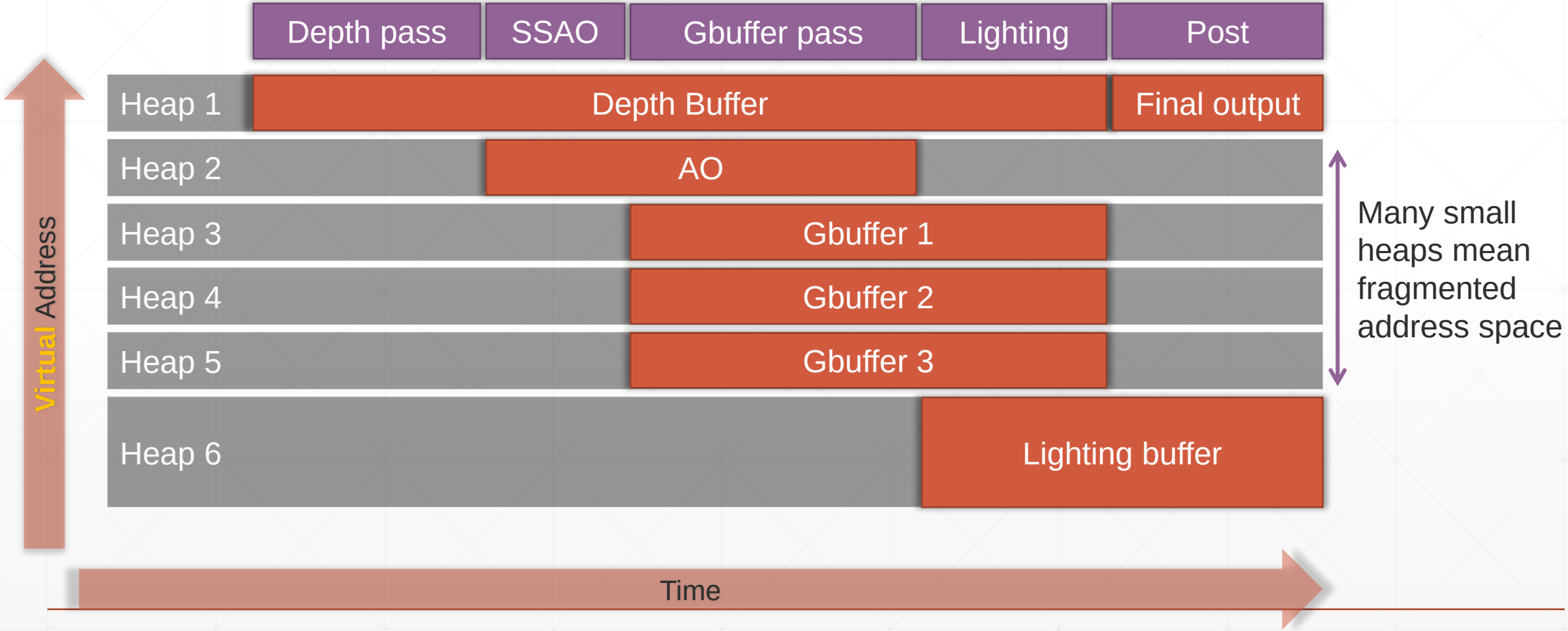


Transient textures on PlayStation 4

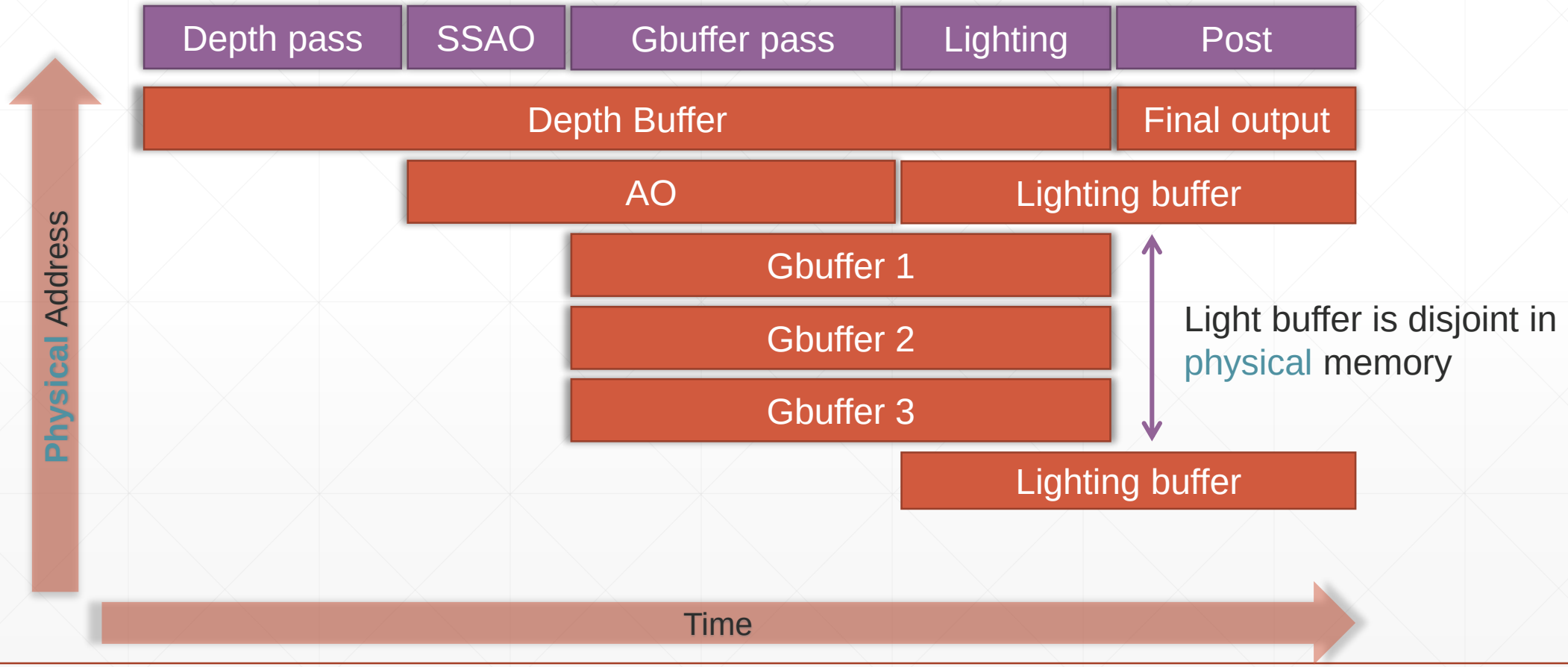


Not globally optimized yet
(heap 2 ▸▸ heap 6)

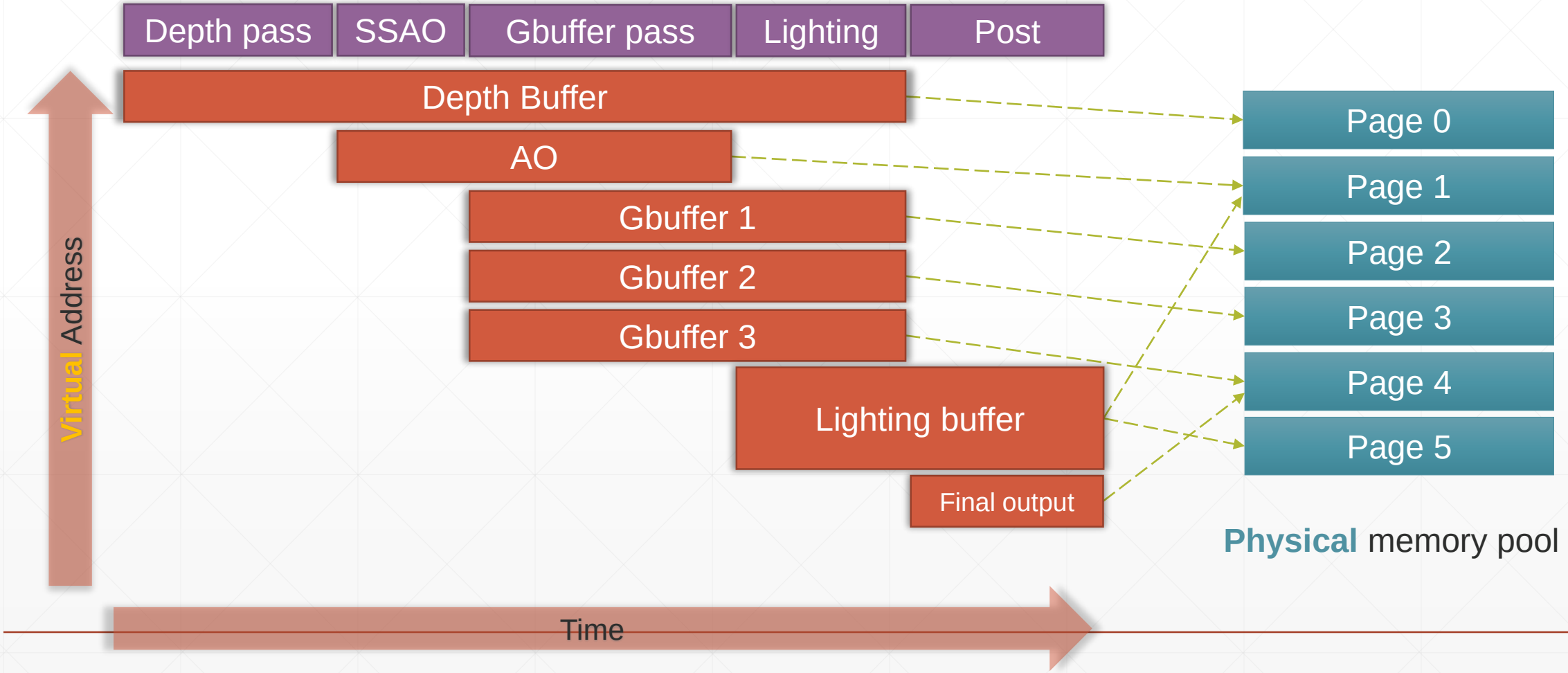
Transient textures on DirectX 12 PC



Transient textures on Xbox One

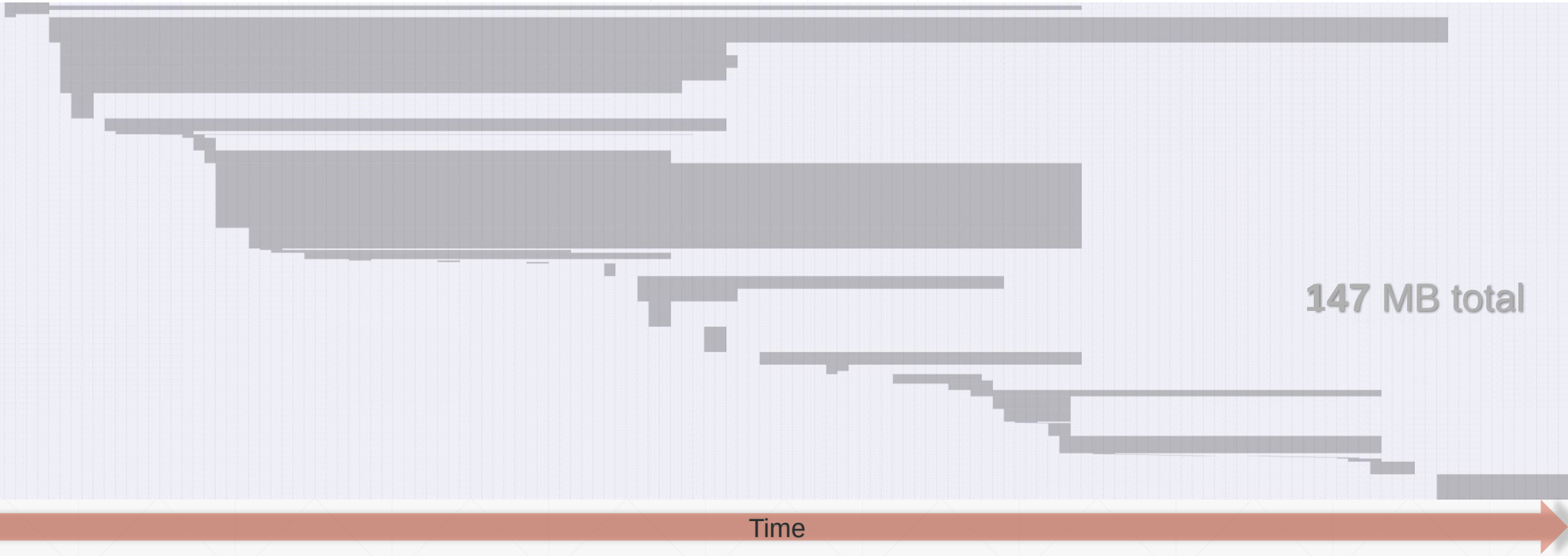


Transient textures on Xbox One

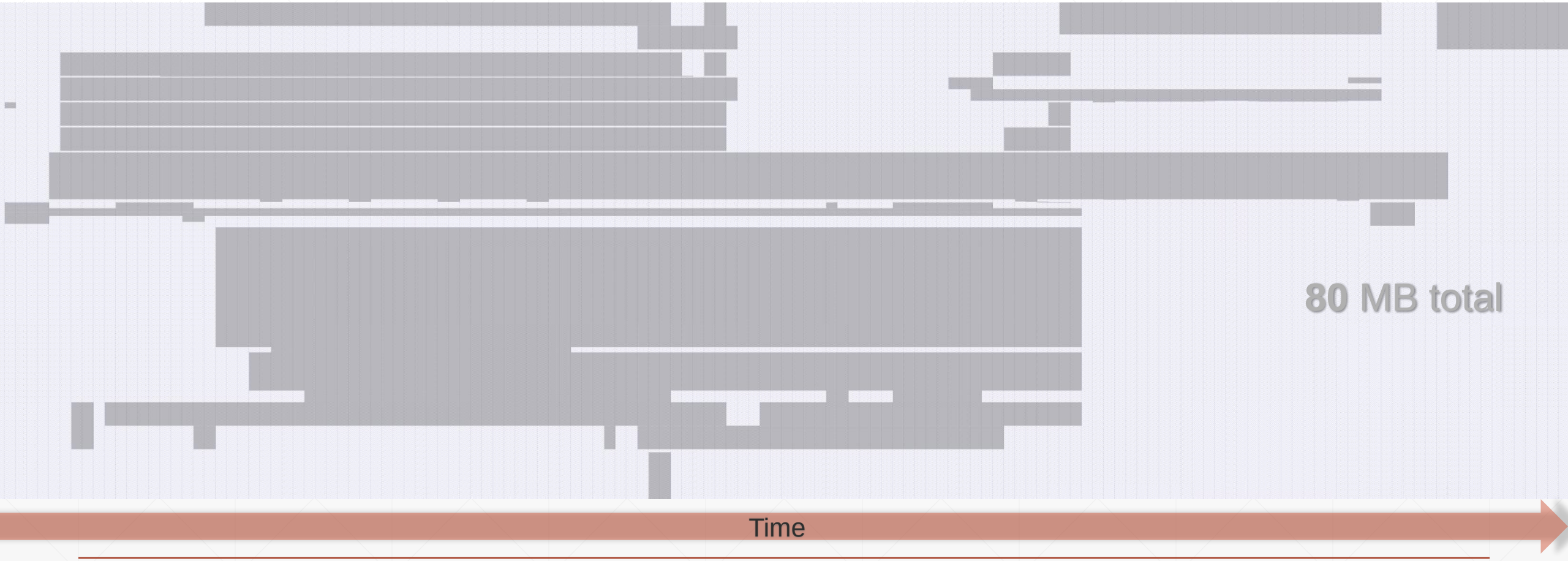



Transient resource allocation results
ENGAGE READY
21/ 84
[AUTO]
3
100

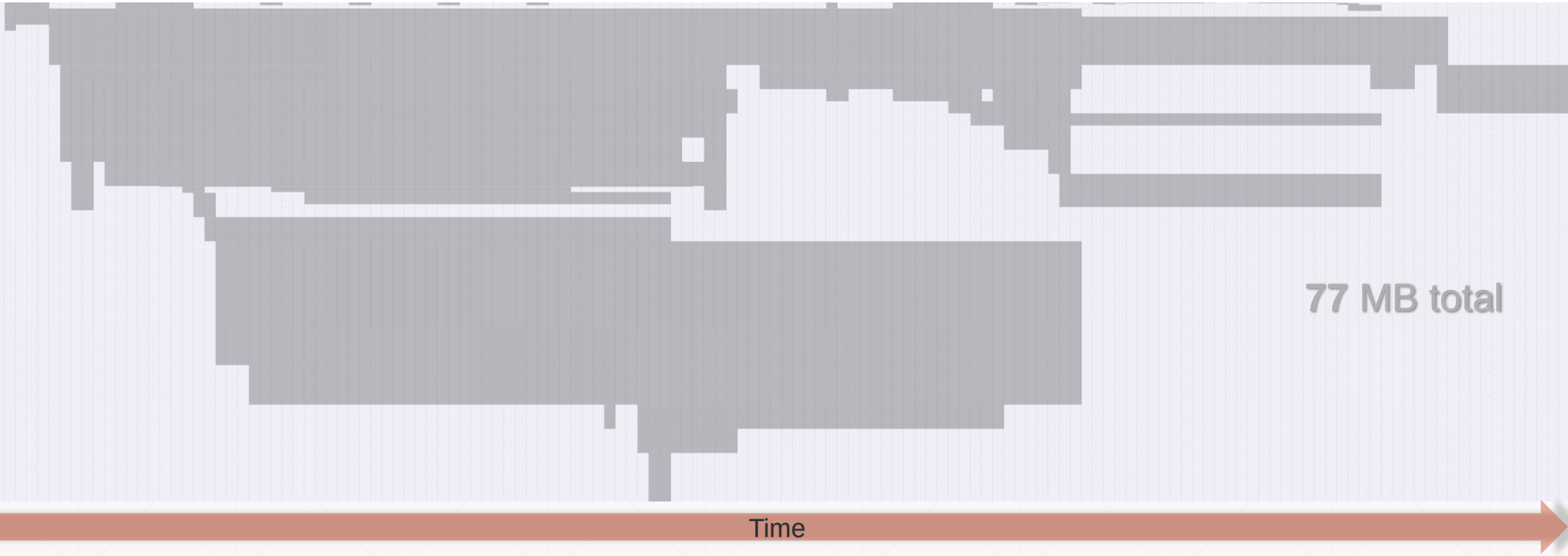
Non-aliasing memory layout (720p)



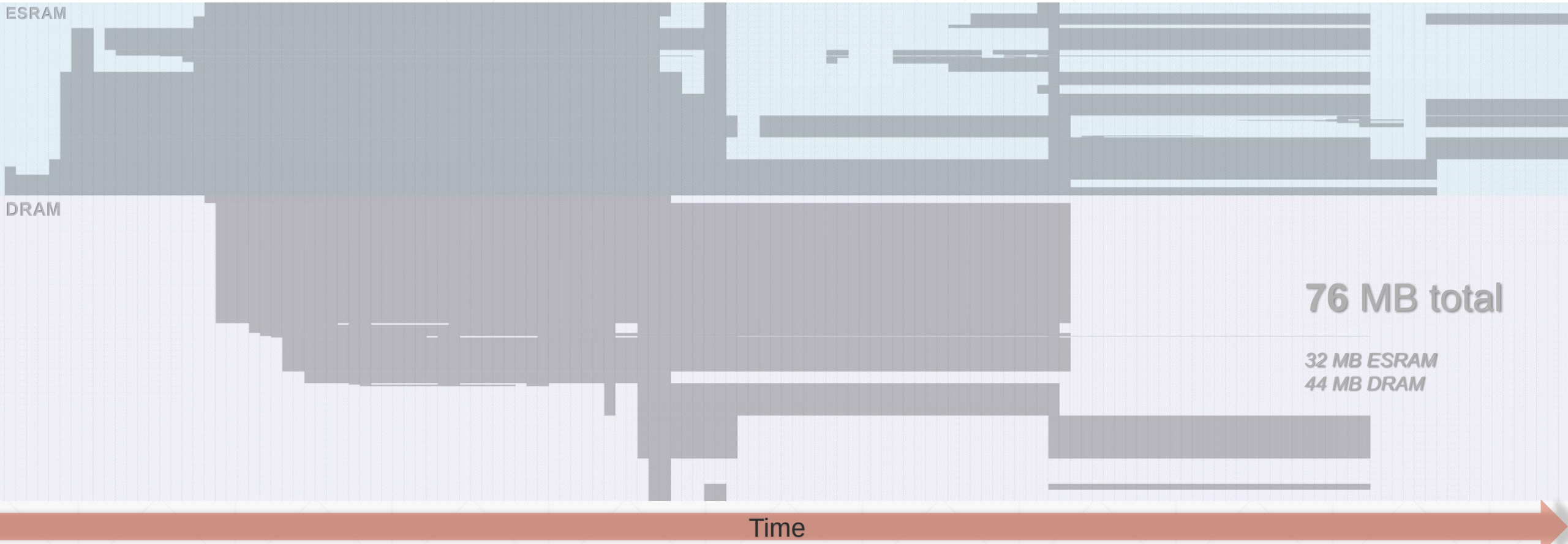
DirectX 12 PC memory layout (720p)



PlayStation 4 memory layout (720p)

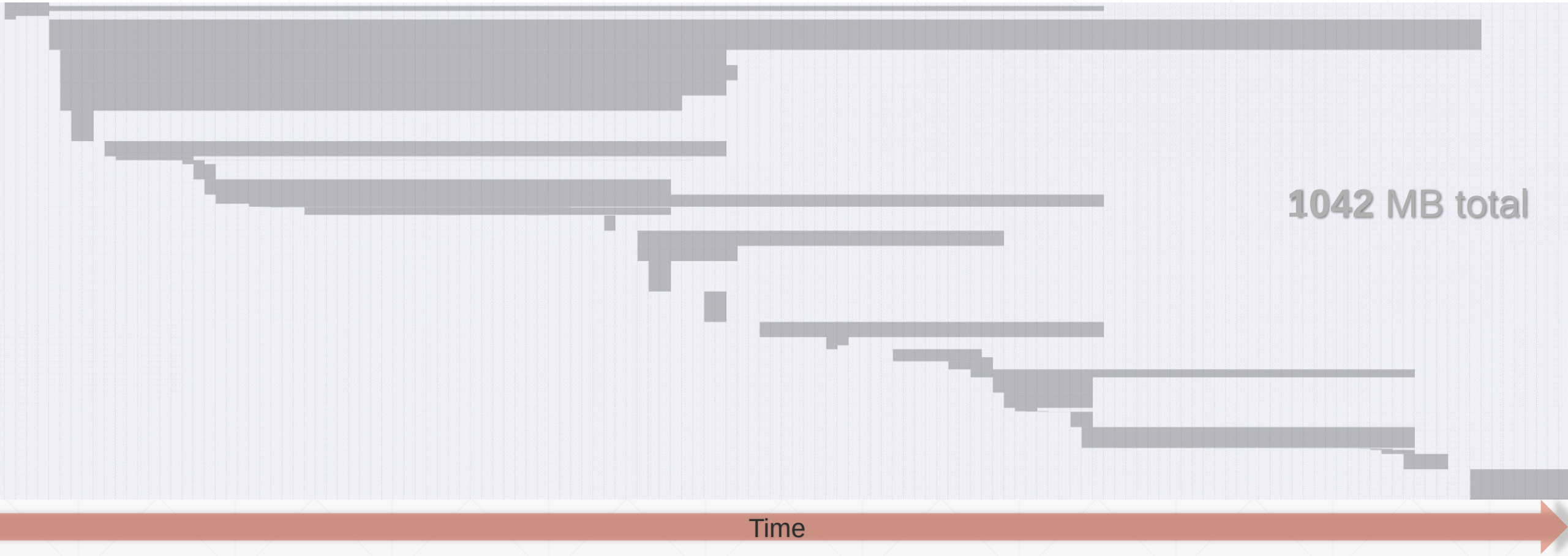


Xbox One memory layout (720p)

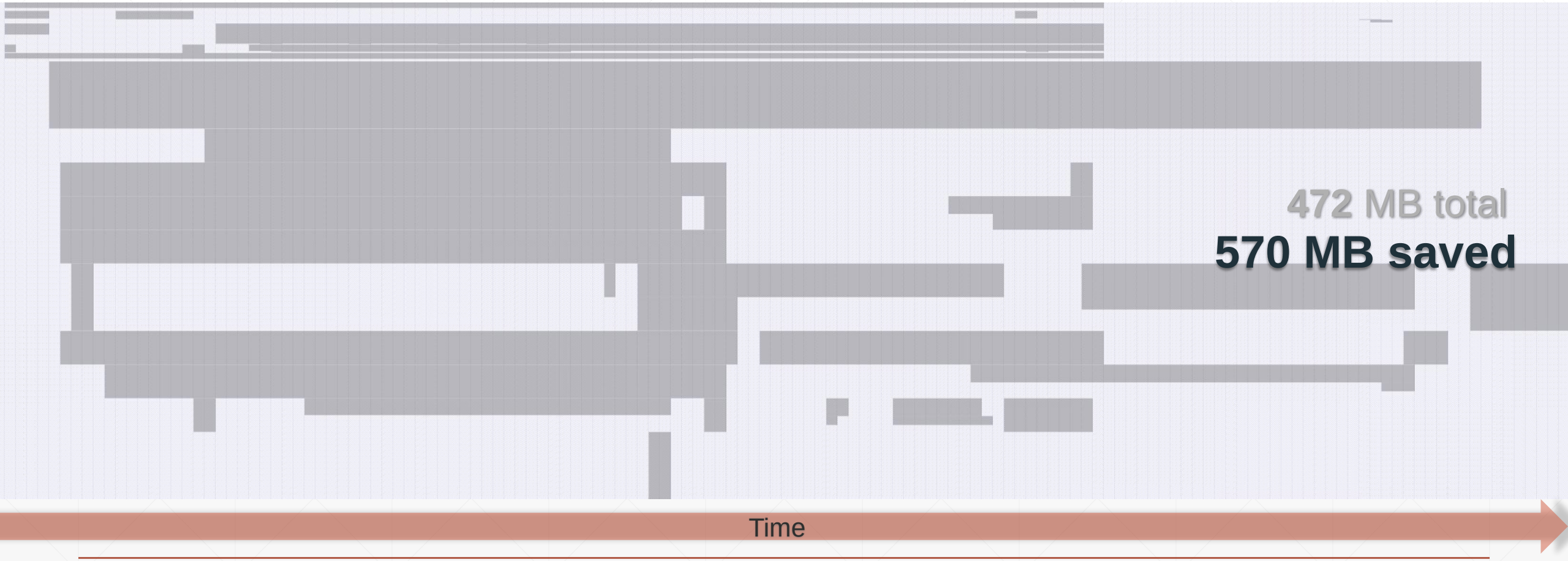


What about 4K?

Non-aliasing memory layout (4K, DX12 PC)



Aliasing memory layout (4K, DX12 PC)



Summary

- Many benefits from full frame knowledge
  - Huge memory savings from resource aliasing
  - Semi-automatic async compute
  - Simplified rendering pipeline configuration
  - Nice visualization and diagnostic tools
- Graphs are an attractive representation of rendering pipelines
  - Intuitive and familiar concept
  - Similar to CPU job graphs or shader graphs
- Modern C++ features ease the pain of retained mode API

Check out “(2017) #gdc2017 FrameGraph - Extensible Rendering Architecture in Frostbite.pptx” for further readings

A. 移动化

Frostbite 面向移动平台的改造

B. 代码实现

围绕数据的改造 (Data-Oriented)

C. 架构改造

FrameGraph 可扩展的渲染架构

内容回顾

Thank you

Compiled by Gu Lu @ Seasun Inc.