GPU Driven Rendering Pipeline или как пишется графика в современных видеоиграх

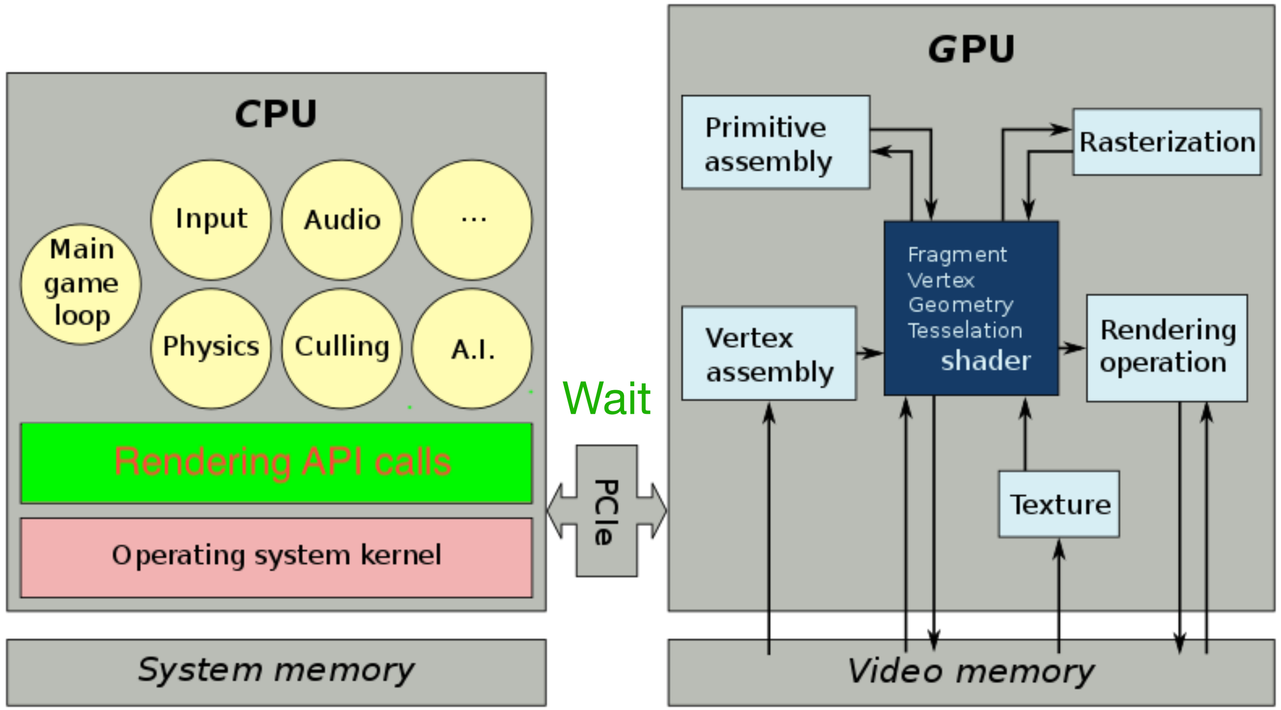
Григорчук Евгений
Elverils

egrigorchuk@elverils.com

C++ Russia

О нас

Game loop игр без GPU Driven pipeline



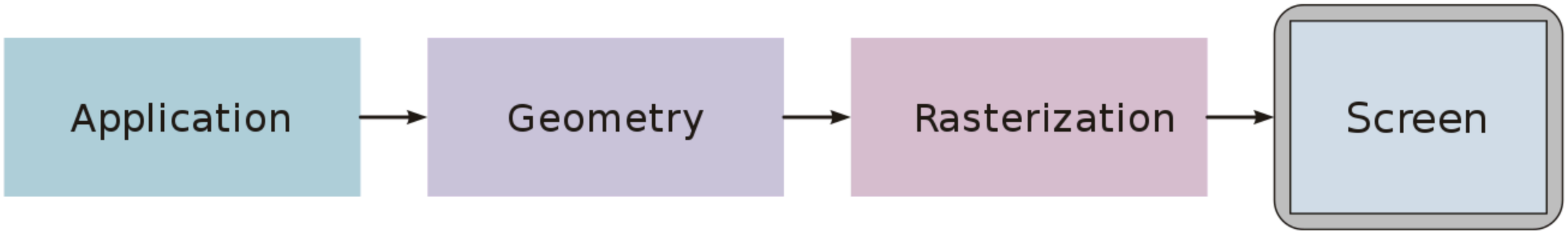
Нужно синхронизировать GPU и CPU

Вычисления физики, cloth, particles на CPU

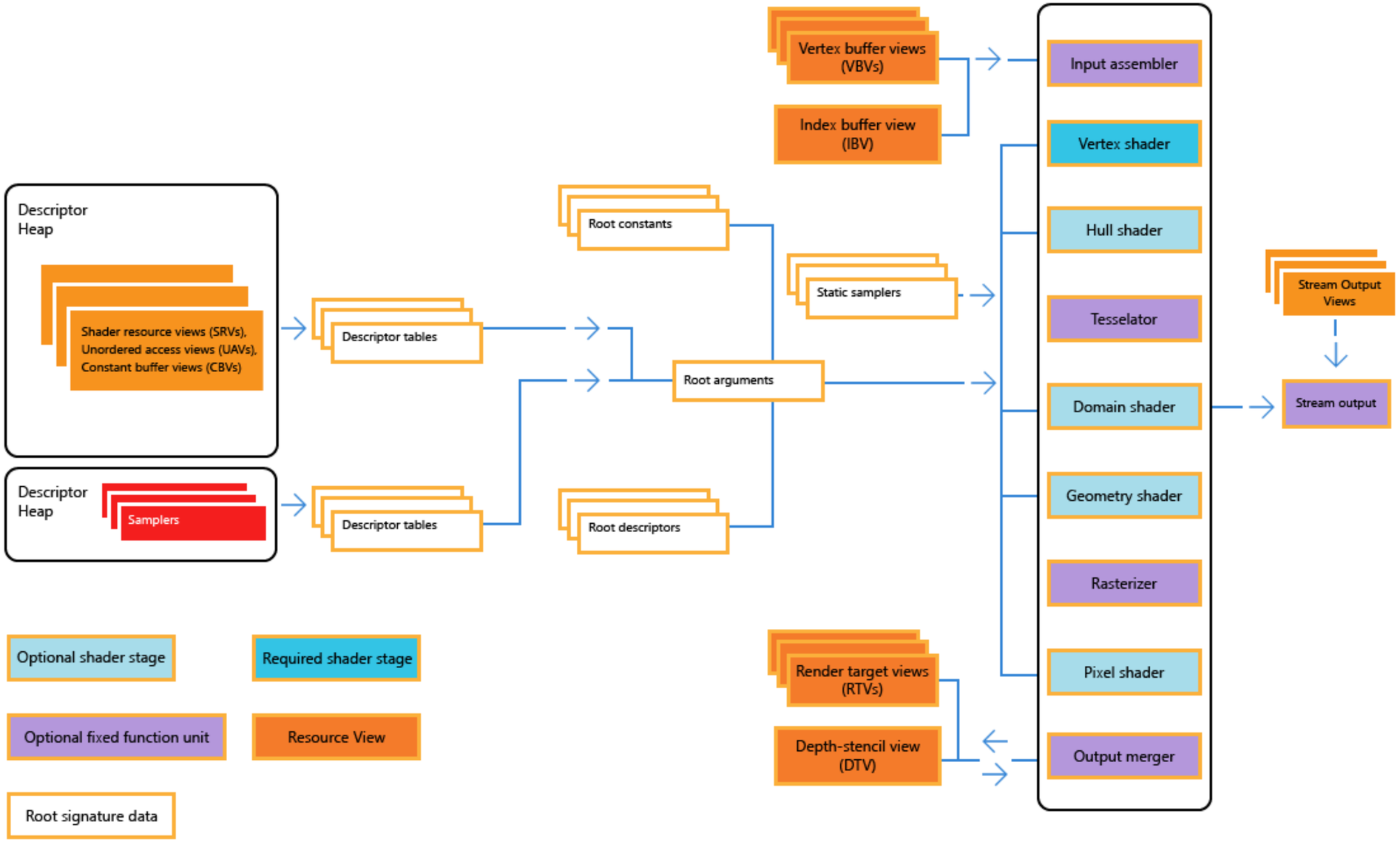
Есть особая лицензия PhysX которая использует GPU calls

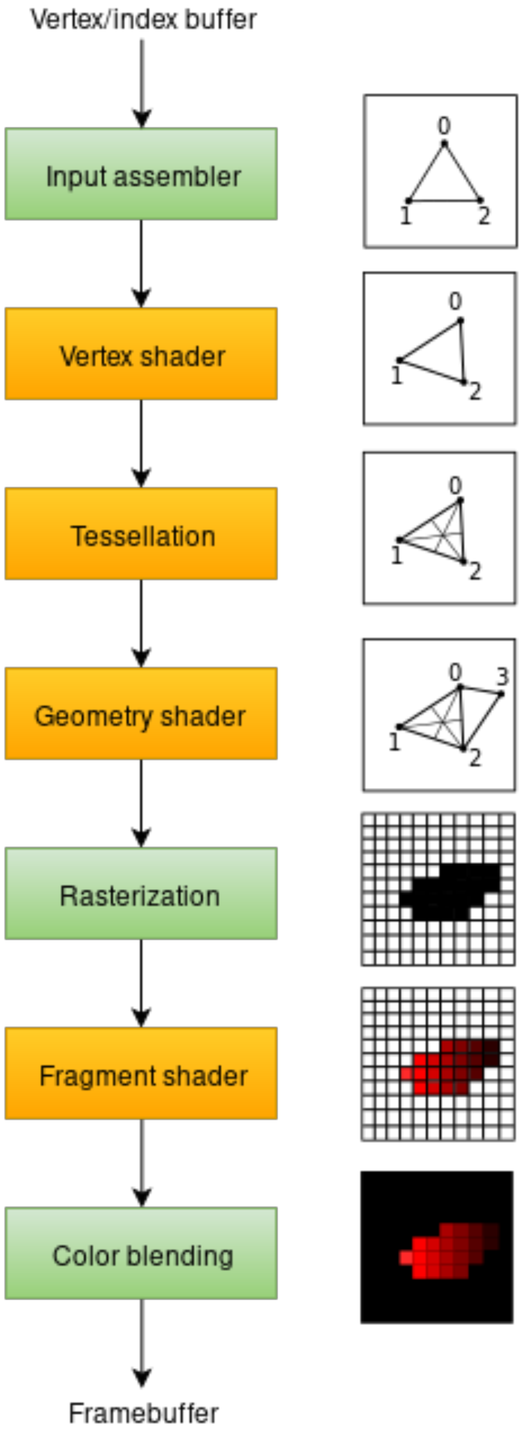
GPU Pipeline

Это концептуальная модель, описывающая шаги, которые должена выполнить GPU для рендеринга/отрисовки 3D-сцены на 2D-экран



GPU Pipeline DX12



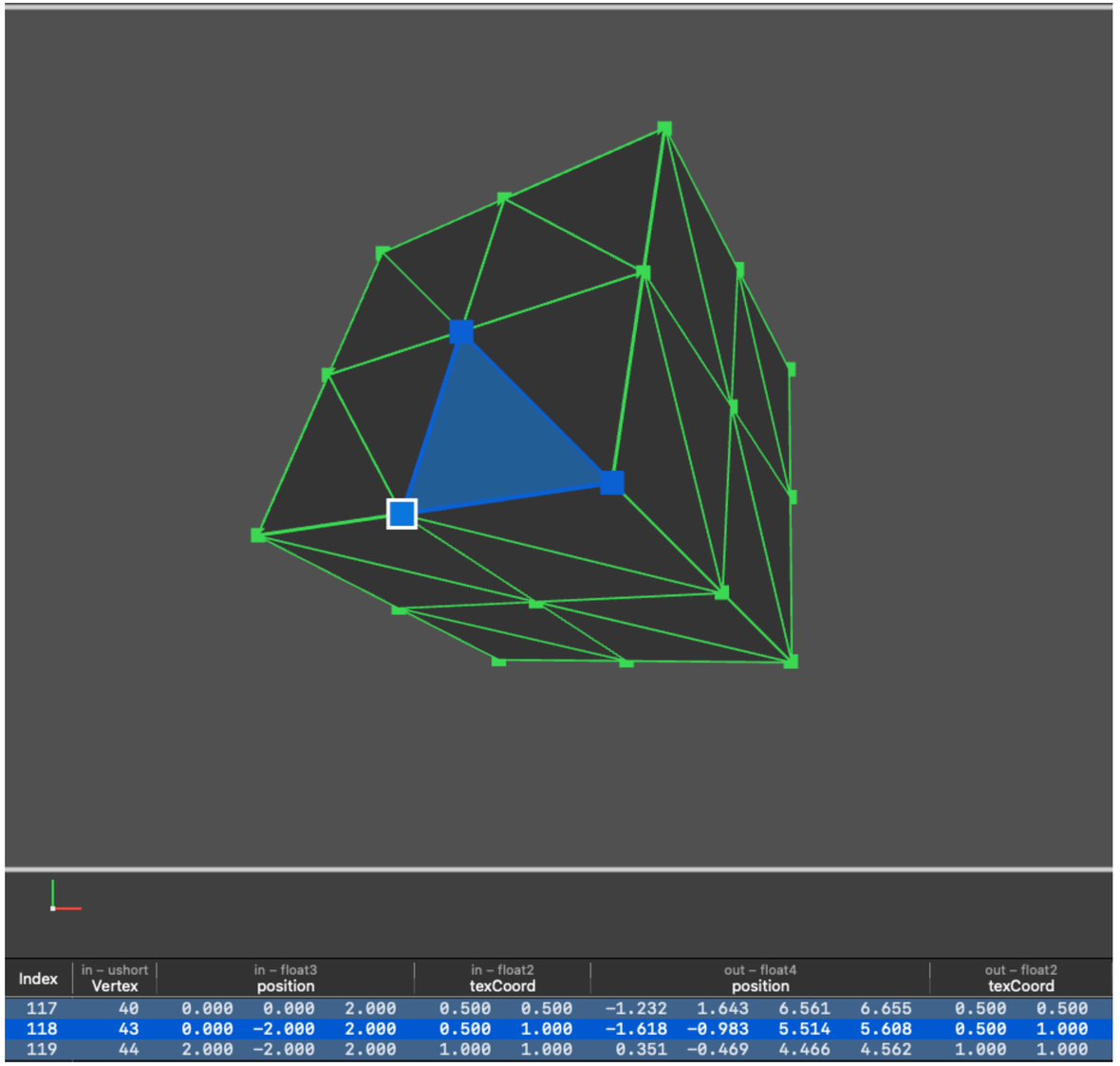


IA – fixed function stage. Использует индексные и вершинные буферы для формирования примитивов и передачи их в последующие этапы

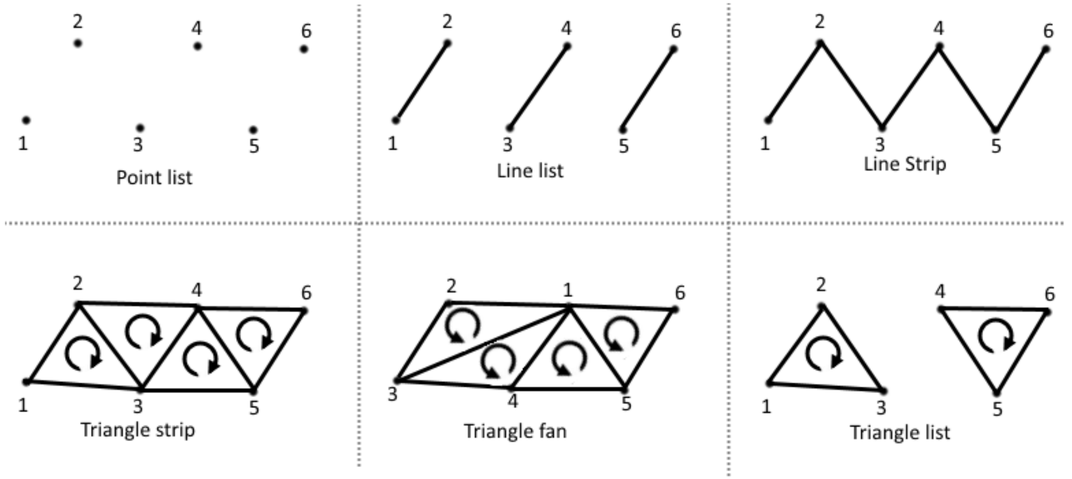
Shader - программа предназначенная для исполнения на GPU

Vertex shader - обрабатывает вершины, обычно выполняя различные преобразования с вершинами. Принимает одну входную вершину и производит одну выходную вершину.

Fragment/Pixel shader - получает интерполированные данные для примитива и формирует данные для пикселя, например цвет



Index	in – ushort Vertex	in – float3 position			in – float2 texCoord		out – float4 position				out – float2 texCoord	
117	40	0.000	0.000	2.000	0.500	0.500	−1.232	1.643	6.561	6.655	0.500	0.500
118	43	0.000	−2.000	2.000	0.500	1.000	−1.618	−0.983	5.514	5.608	0.500	1.000
119	44	2.000	−2.000	2.000	1.000	1.000	0.351	−0.469	4.466	4.562	1.000	1.000

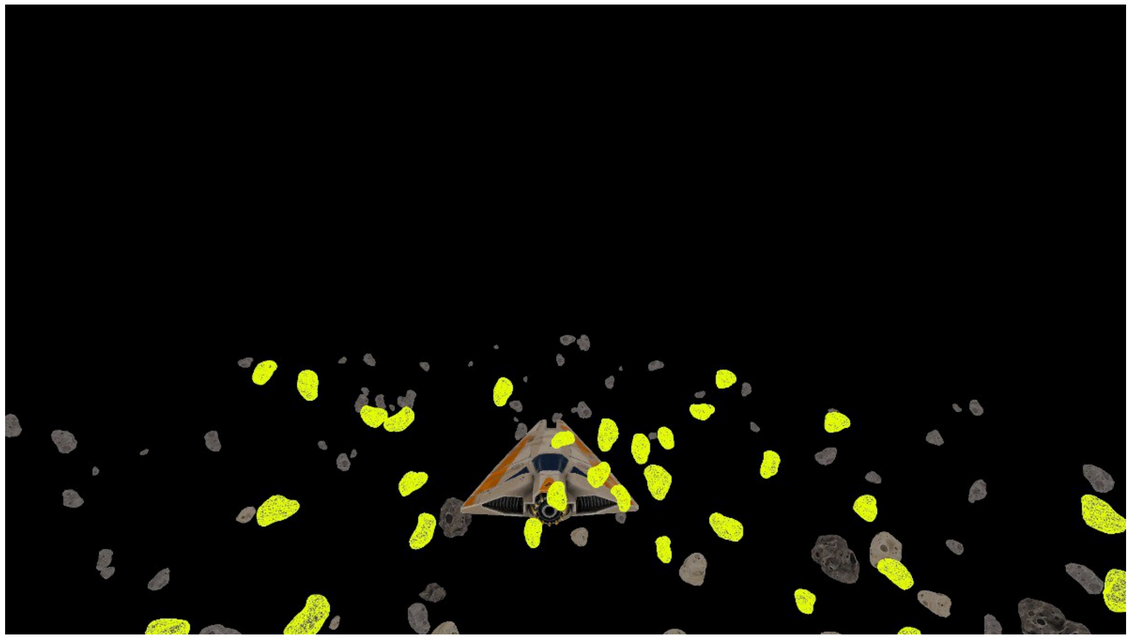


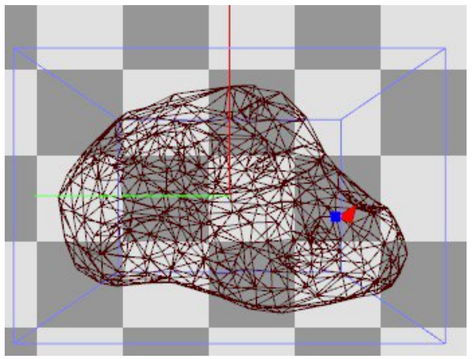
Причины использования треугольников

- Эффективно используют память
- Можно представить другую фигуру
- Планарные

Улучшение графики в играх

Instancing





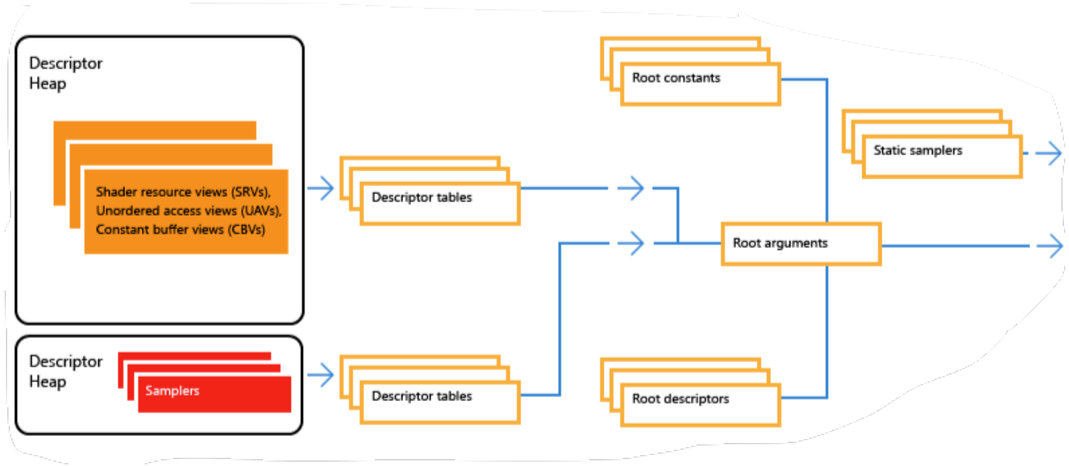
Это техника при которой мы рисуем много объектов (с одинаковыми мешами) одновременно одним вызовом drawcall, что экономит нам все общение между CPU и GPU.

Нам нужно дополнительно передавать два параметра

1) instance count - количество объектов которые мы хотим нарисовать

2) instance buffer - содержит в себе данные которые отличаются в объектах

Что API научились оптимизировать



DX12 -> PSO (Pipeline State Object)
Vulkan -> Pipeline
Metal -> Pipeline state

В крупных проектах за оптимизацию по созданию pipeline state может отвечать несколько человек

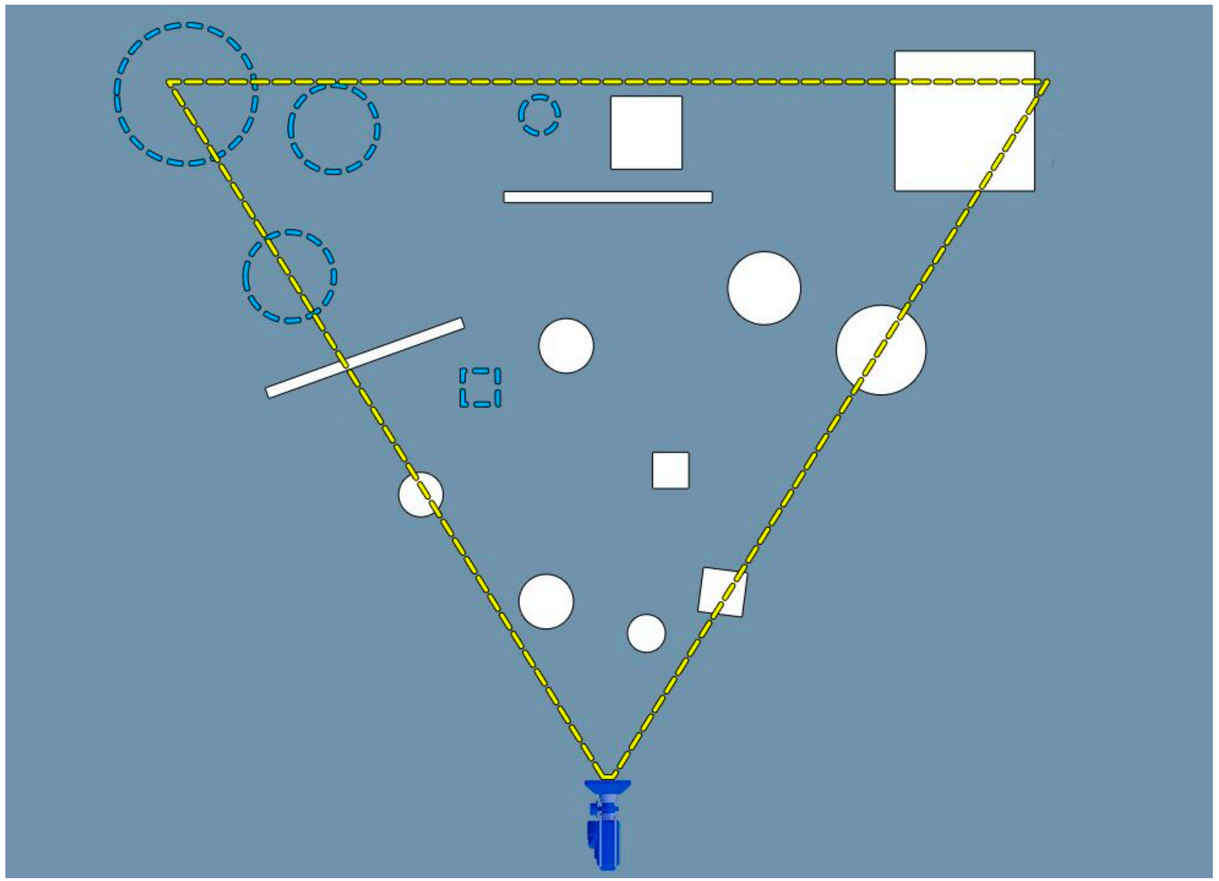
GPU Driven pipeline

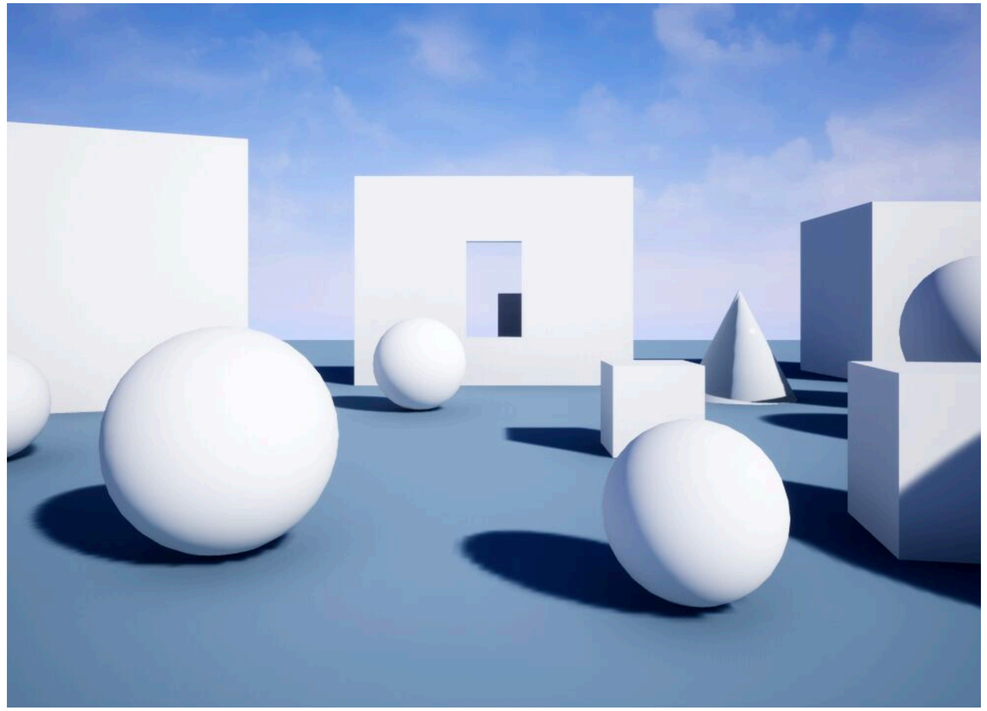
Причина: GPU мощнее чем CPU (игровые консоли, apple silicon)

Как написать игру с поддержкой GPU driven pipeline?

Culling

Отбраковка объектов которые не попадают в кадр, перед рисованием





Indirect

Позволяет настраивать функции drawcall непосредственно на GPU
Это позволило перенести culling/particle/cloth/water ... с CPU на GPU

```
typedef struct {
    uint32_t vertexCount;
    uint32_t instanceCount;
    uint32_t vertexStart;
    uint32_t baseInstance;
} MTLDrawPrimitivesIndirectArguments;
```

Indirect

```
[renderEncoder executeCommandsInBuffer:_indirectCommandBuffer withRange:NSMakeRange(0, 128)];
```

```
[renderEncoder drawPrimitives:MTLPrimitiveTypeTriangle
                  vertexStart:0
                  vertexCount:vertexCount
                instanceCount:1
                 baseInstance:objIndex];
```

x 16

MTLIndirectCommandBuffer может хранить в себе draw команды, что позволяет его использовать несколько раз в свою очередь как MTLCommandBuffer является одноразовым

Bindless and binding models

Binding model

Ставим текстуры с CPU для каждого GPU pass

Недостатки:

- Слоты доступны в ограниченном количестве.

- Недостаточная гибкость в настройке input

- CPU cost на привязку дескрипторов, что является основной причиной проблем с производительностью CPU в коде рендеринга

Bindless and binding models

Binding model

DX

```
Shader:
Texture2D<float4> texture : register(t27);
float4 color = texture.Sample(sampler_linear_clamp, uv);

CPU code:
Texture texture;
device->BindResource(PS, &texture, 27, cmd);
```

Metal

```
Shader:
texture2D<float> texture [[ texture(27) ]]
float4 color = texture.sample(sampler_linear_clamp, uv);

CPU code:
id<MTLTexture> texture;
[renderEncoder setFragmentTexture: texture atIndex: 27]
```

Bindless and binding models

Bindless model

Вместо бинда для каждого pass обращаемся к текстурам как к массиву данных

DX 12

```
struct Material
{
  int texture_basecolormap_index;
  int texture_normalmap_index;
  int texture_emissivemap_index;
};
```

Metal

```
struct Material
{
  texture2d<float> texture_basecolormap_index [[ id(0) ]];
  texture2d<float> texture_normalmap_index    [[ id(1) ]];
  texture2d<float> texture_emissivemap_index  [[ id(2) ]];
};
```

Argument buffers

```
struct My_AB {
    texture2d<float, access::write> a;
    depth2d<float> b;
    sampler c;
    texture2d<float> d;
    device float4* e;
    texture2d<float> f;
    int g;
};
kernel void my_kernel(constant My_AB & my_AB [[buffer(0)]])
{ ... }
```

Полезный совет - выделяйте память для argBuffer из heap, это снизит затраты на отслеживание ресурсов

ArgBuffer позволяет динамически индексировать ресурсы во время выполнения функции это позволяет снизить лимит для входных параметров в шейдерах.

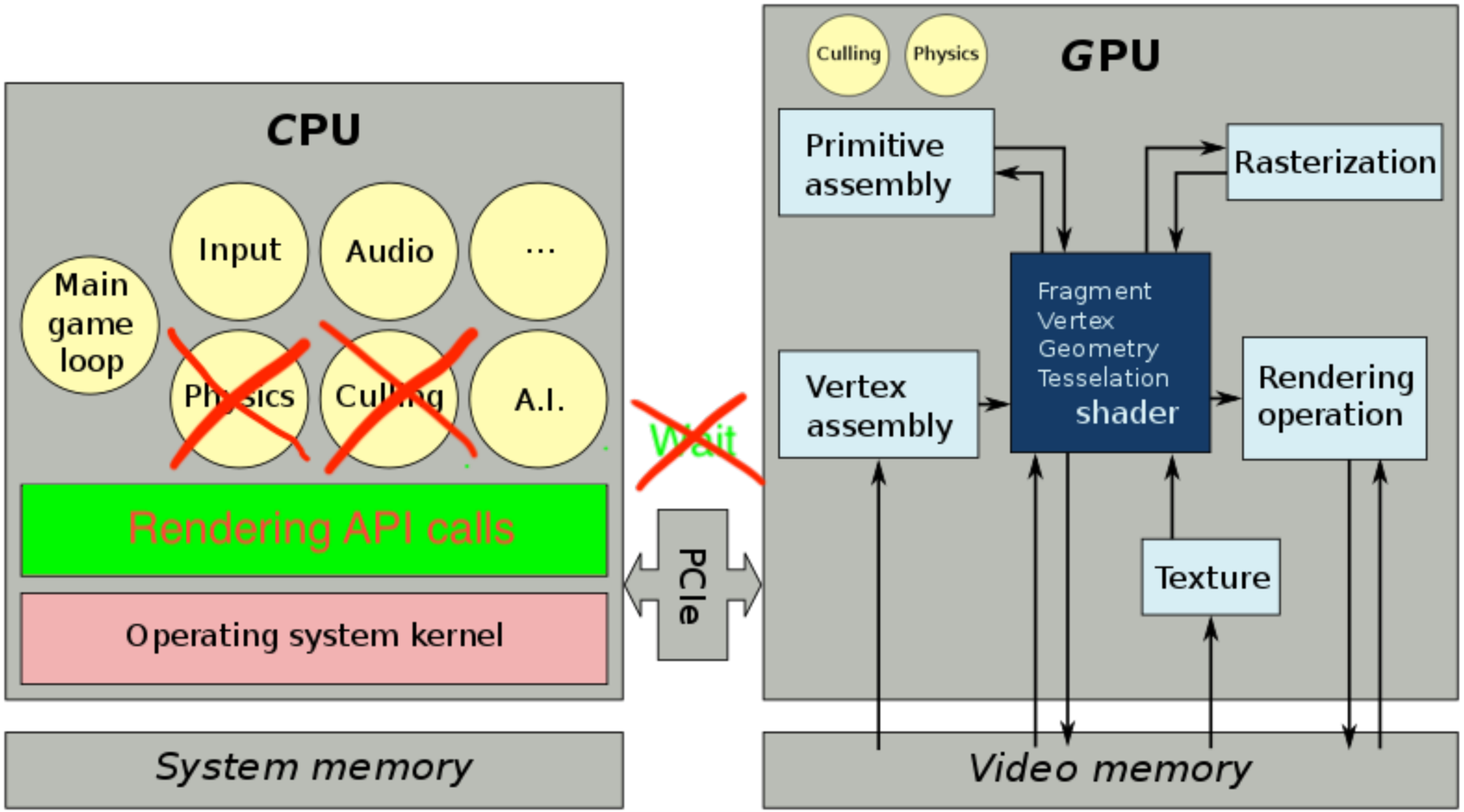
Мы можем использовать его многократно

```
// The argument buffer that defines materials and particle properties
struct TerrainHabitat
{
private:
    TerrainHabitat ();
public:

    float slopeStrength      [[ id(TerrainHabitat_MemberIds::slopeStrength)      ]];
    float slopeThreshold     [[ id(TerrainHabitat_MemberIds::slopeThreshold)     ]];
    float elevationStrength  [[ id(TerrainHabitat_MemberIds::elevationStrength)  ]];
    float elevationThreshold [[ id(TerrainHabitat_MemberIds::elevationThreshold) ]];
    float specularPower      [[ id(TerrainHabitat_MemberIds::specularPower)      ]];
    float textureScale       [[ id(TerrainHabitat_MemberIds::textureScale)       ]];
    bool  flipNormal         [[ id(TerrainHabitat_MemberIds::flipNormal)         ]];

    struct ParticleProperties
    {
        // The fields of this struct must be reflected in TerrainHabitat_MemberIds
        simd::float4    keyTimePoints;
        simd::float4    scaleFactors;
        simd::float4    alphaFactors;
        simd::float4    gravity;
        simd::float4    lightingCoefficients;
        int             doesCollide;
        int             doesRotate;
        int             castShadows;
        int             distanceDependent;
    } particleProperties;

    texture2d_array <float,access::sample> diffSpecTextureArray [[ id(TerrainHabitat_MemberIds::diffSpecTextureArray) ]];
    texture2d_array <float,access::sample> normalTextureArray   [[ id(TerrainHabitat_MemberIds::normalTextureArray)   ]];
};
```

Как выглядит код на GPU

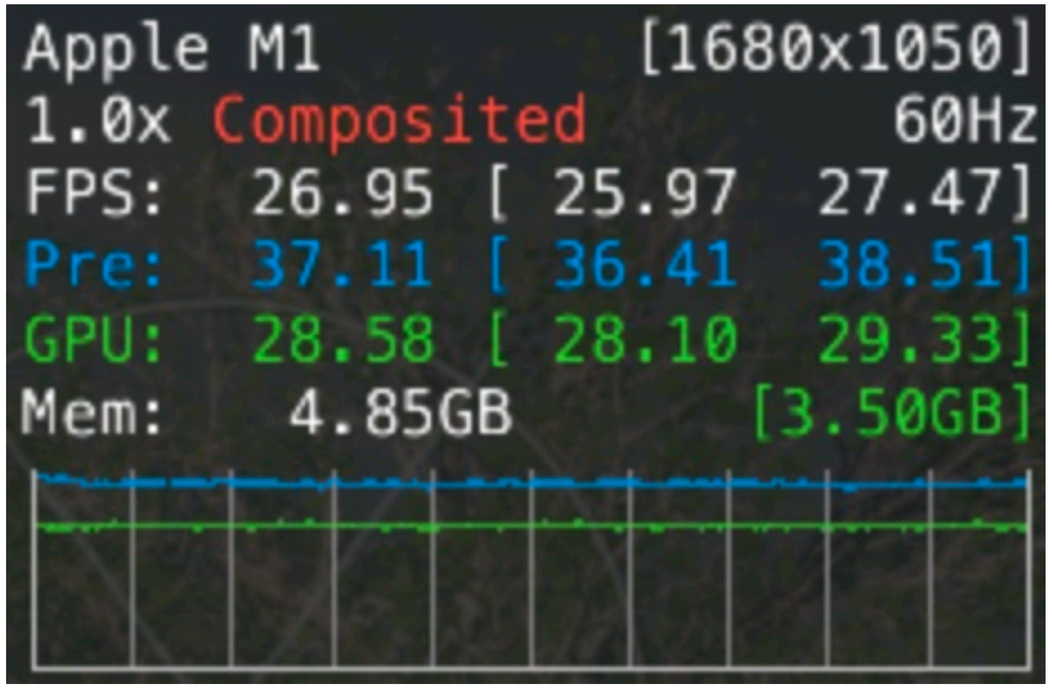
```
struct ICBContainer
{
    command_buffer commandBuffer [[ id(0) ]];
};

kernel void
cullMeshesAndEncodeCommands(uint                                objectIndex   [[ thread_position_in_grid ]],
                            constant AAPLFrameState     *frame_state   [[ buffer(0) ]],
                            device AAPLObjectPerameters *object_params [[ buffer(1) ]],
                            device AAPLVertex           *vertices      [[ buffer(2) ]],
                            device ICBContainer         *icb_container [[ buffer(3) ]])
{

  ...

  if(visible)
   {
       render_command cmd(icb_container->commandBuffer, objectIndex);

        cmd.set_vertex_buffer(frame_state,   AAPLVertexBufferIndexFrameState);
        cmd.set_vertex_buffer(object_params, AAPLVertexBufferIndexObjectParams);
        cmd.set_vertex_buffer(vertices,      AAPLVertexBufferIndexVertices);

        cmd.draw_primitives(primitive_type::triangle,
                            object_params[objectIndex].startVertex,
                            object_params[objectIndex].numVertices, 1,
                            objectIndex);
```

Game loop игр с GPU Driven pipeline

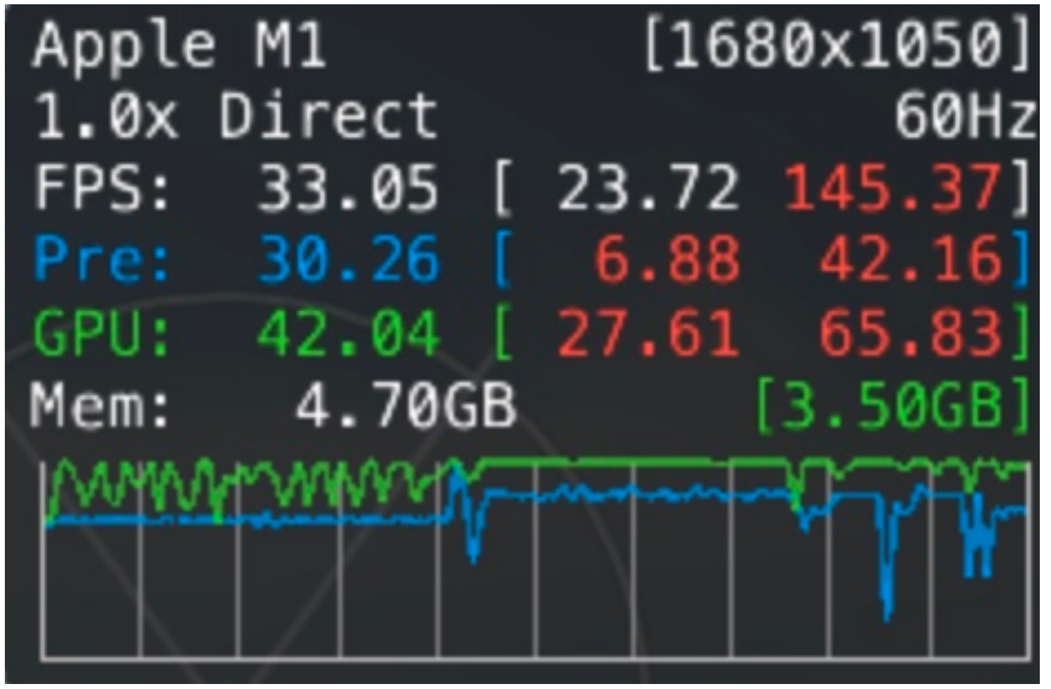


Прирост производительности

GPU Driven pipeline - выключен



GPU Driven pipeline - включен



GPU Driven pipeline итоги

- GPU culling
- Минимизировать переключение GPU pipeline state
- Использовать один большой буффер со всеми мешами
- Использовать bindless модель
- Indirect Command Encoding
- Генерация всех render commands на GPU (Metal only)

Elverils

code magic at our fingertips

egrigorchuk@elverils.com

info@elverils.com