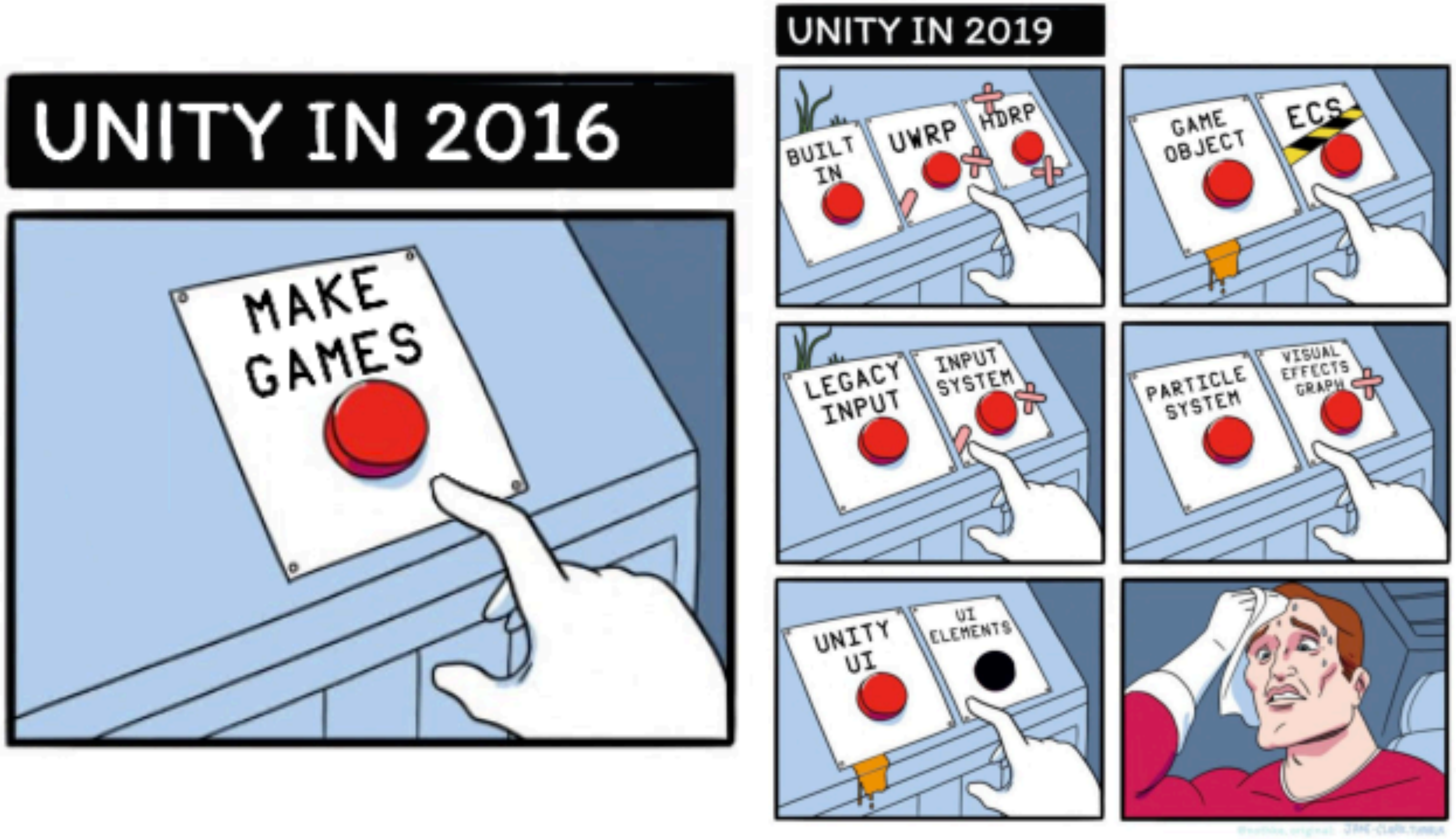
Graphics in Unity
Part1

Data Vis @ KADK
Sept-Oct 2020

Carl Emil Carlsen
Artist, designer, teacher.
twitter.com/cecarlsen | cec.dk

State of graphics in Unity
The shader pipeline



By @ SalahChafai160
https://forum.unity.com/threads/future-of-unity.892993/

Render Pipeline Comparison
2019.3

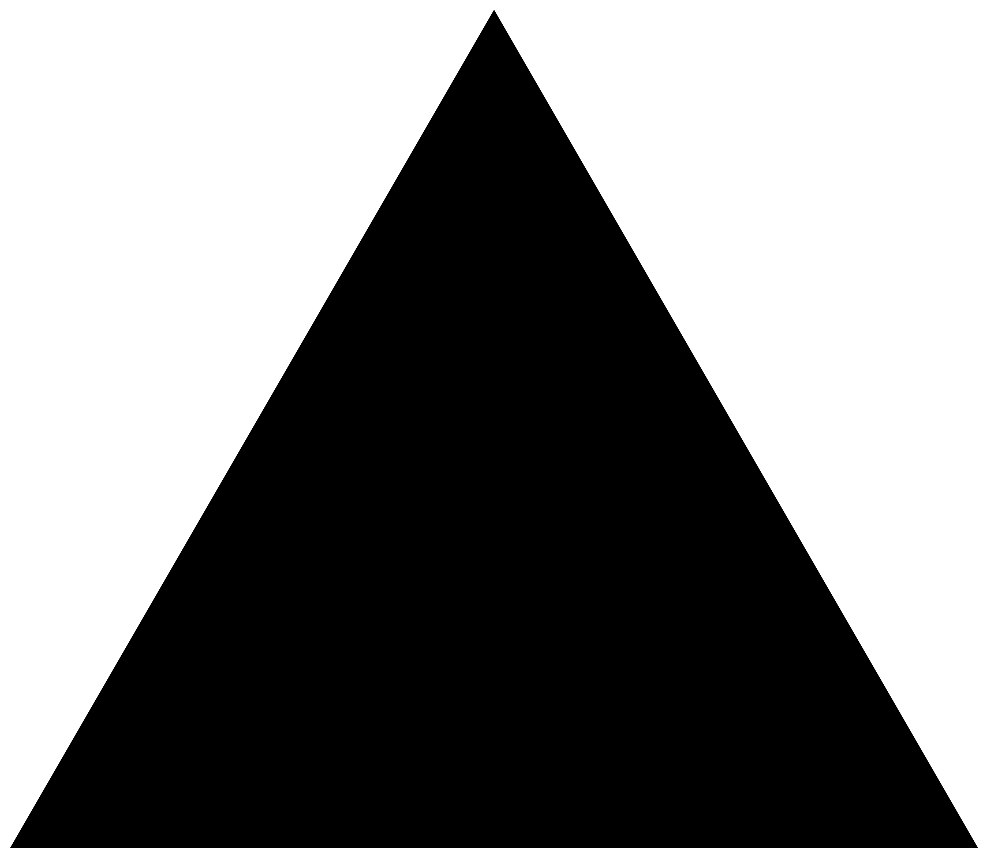
	Built-In Render Pipeline	Universal Render Pipeline	High-Definition Render Pipeline
Platform Coverage	• All platforms	• Mobile • XR • PC & Consoles, incl. Nintendo Switch	• PC & Consoles (excl. Nintendo Switch) • Metal • Vulkan • XR
Rendering Path	• Multi-pass Forward • Multi-pass Deferred	• Forward • 2D renderer	• Deferred • Forward • Both Forward and Deferred
Color Space	• Linear with sRGB light intensity • Gamma	• Linear with linear light intensity • Gamma (for legacy hardware)	• Linear with linear light intensity
HDR Support	• Yes		
Anti-Aliasing	• MSAA (Forward) • FXAA • TAA • SMAA	• MSAA • FXAA • SMAA	• MSAA (Forward) • TAA • FXAA • SMAA
Light entities	• Non-physically based intensity unit • Non-physically based falloff • Realtime Directional, Spot and Point lights • Area light ◦ Realtime rectangle • Max # lights in forward per pixel: 8 • Max # lights in deferred per pixel: unlimited • Max # lights per vertex: 4	• Non-physically based intensity unit • Physically based falloff • Realtime Directional, Spot, Point lights • Baked Rectangle and Disk Area Lights • Max # lights: 1 Direction. + 8 (4 on gles2) lights • Max # lights per camera: 256 (32 on mobile)	• Physically based intensity units • Physically based falloff • Realtime Directional, Spot and Point lights • Spot light types ◦ Cone ◦ Pyramid ◦ Box • Realtime Rectangle Area light • Realtime Line light • Max # lights per pixel: 24
Realtime Shadow Casting Lights	• Directional • Spot • Point	• Directional • Spot	• Directional • Spot • Point • Area (Rectangle)
Light Modes	• Baked • Mixed ◦ Subtractive ◦ Baked Indirect ◦ Shadowmask Mode: Shadowmask ◦ Shadowmask Mode: Distance Shadowmask • Realtime	• Baked • Mixed ◦ Subtractive ◦ Baked Indirect • Realtime	• Baked • Mixed ◦ Baked Indirect ◦ Hybrid Shadowmask • Realtime
Shaders	• Unified PBS Shader ◦ Metallic workflow ◦ Specular workflow • Many non-PBS shaders	• Unified PBS Shader ◦ Metallic workflow ◦ Specular workflow • Unified non-PBS Shader ◦ Covers all non-PBS shaders from Built-in RP • Shader Graph	• Advanced Unified PBS Shaders ◦ Metallic workflow ◦ Specular workflow • Shader Graph
Global Illumination	• Baked Lightmap • Baked Light Probes • Realtime Light Probes • Realtime GI	• Baked Lightmap • Baked Light Probes • Realtime GI Not Supported ⚠️	• Baked Lightmap • Baked Light Probes • Realtime GI Not Supported ⚠️
Motion Vectors	• Yes	• None	• Motion Vectors • Distortion Vectors
Post-Processing	• Post-Processing Stack V2 • No Object Motion Blur	• Native post-effects • No Object Motion Blur • No Ambient Occlusion • No Exposure	• Native post-effects
Sky lighting	• Procedural Sky • Cubemap/LatLong Sky • Ambient Lighting	• Procedural Sky • Cubemap • Ambient Lighting	• Dedicated sky manager system, no ambient light • GGX GPU convolution of the sky • Procedural sky (from Built-In RP) • PBR Sky • HDRI Sky
Fog	• Linear • Exponential • Exponential Squared		• Exponential • Volumetric
Ray-tracing	• No		• Yes

https://docs.unity3d.com/2020.1/Documentation/Manual/BestPracticeLightingPipelines.html

State of drawing APIs in Unity

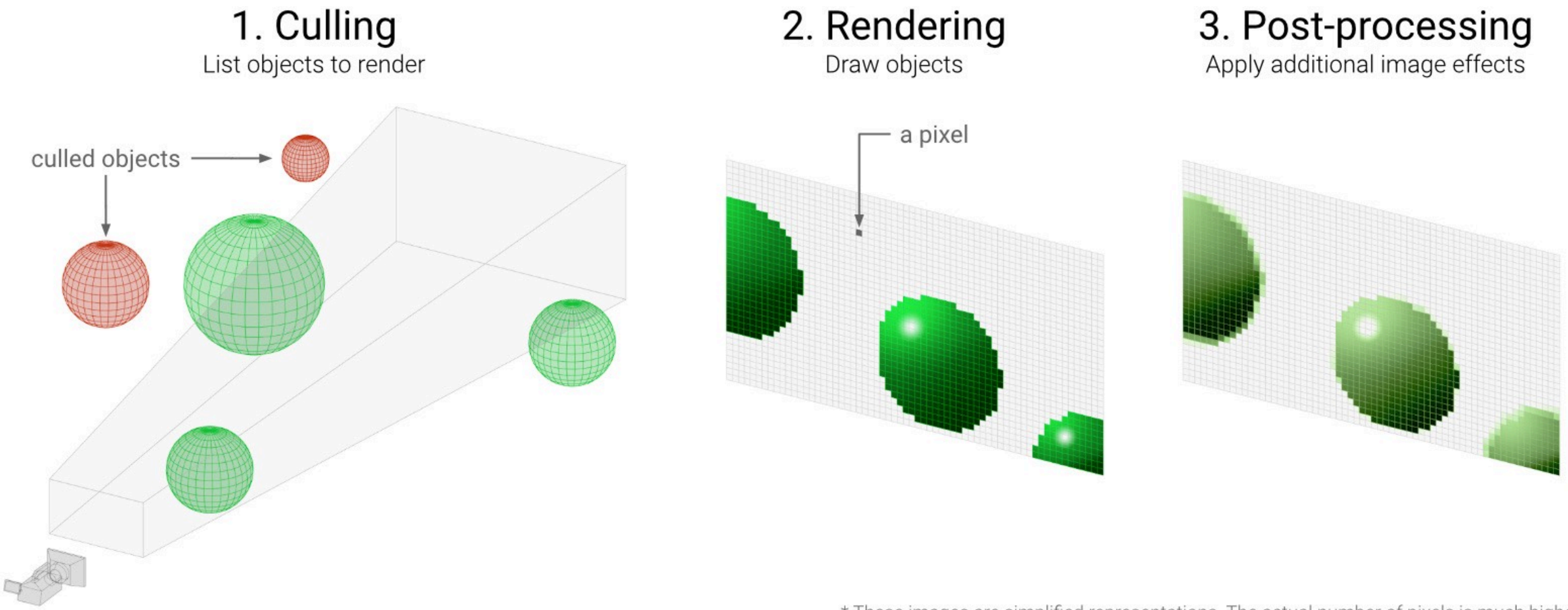
API	Data read	Built-In RP	URP	HDRP	
GL Class	CPU	Yes	As overlay	No	
Mesh	CPU	Yes	Yes	Yes	
Unity UI (Canvas)	CPU	Yes	Yes	Yes	To be deprecated
UI Toolkit (UIElements)	CPU	Soon	Soon	Soon	To replace Unity UI
Graphics.DrawProcedural	GPU	Yes	No	No	
Custom Pass (DrawProcedural)	GPU	No	Yes	Yes	Not the same for each SRPs and overly complex.
VFX Graph	GPU	No	Soon	Yes	Limited capability
DOTS + Mesh	CPU	Yes	Yes	Yes	High complexity

How to draw a triangle



https://github.com/cecarlsen/HowToDrawATriangle

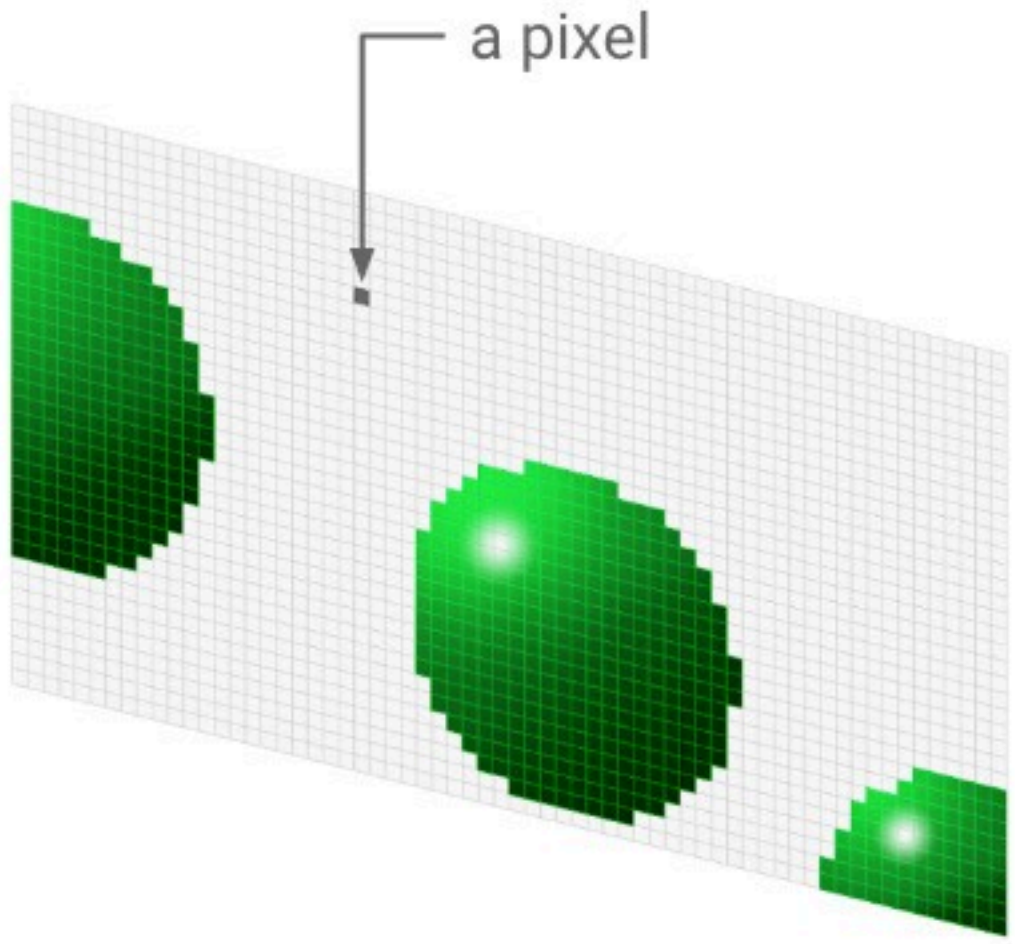
Rendering process



* These images are simplified representations. The actual number of pixels is much higher.

1. Culling
List objects to render

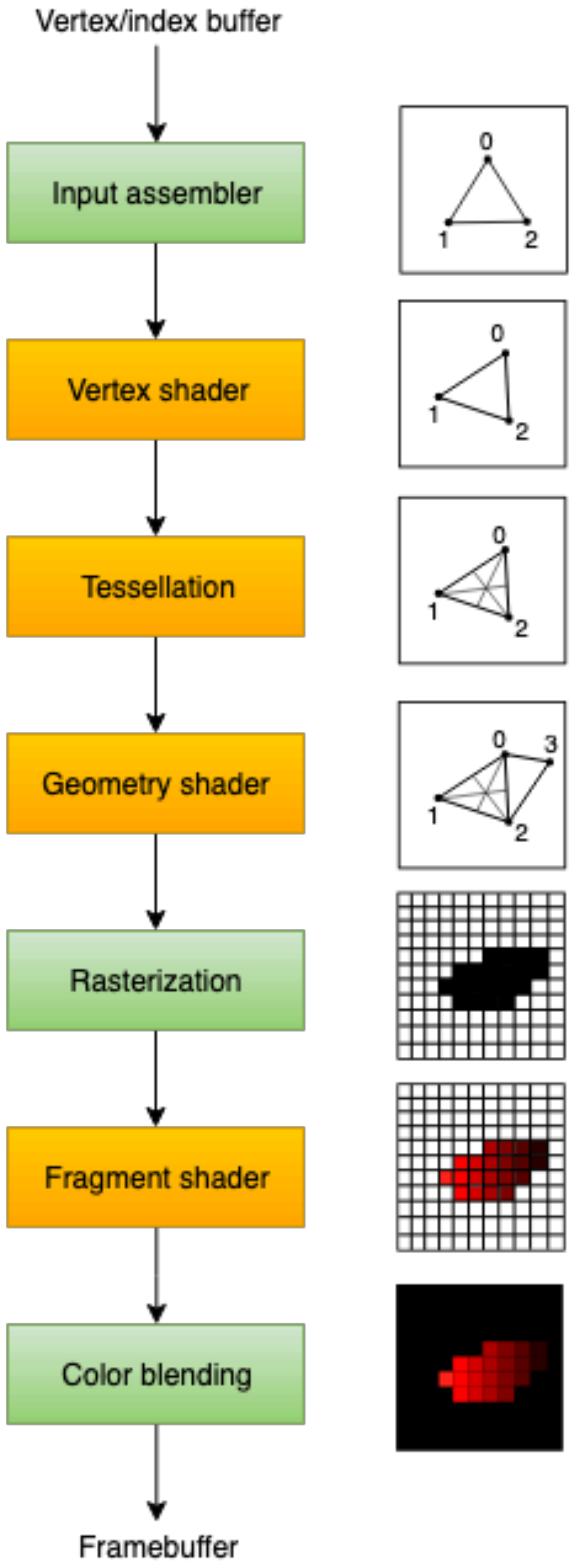
2. Rendering
Draw objects

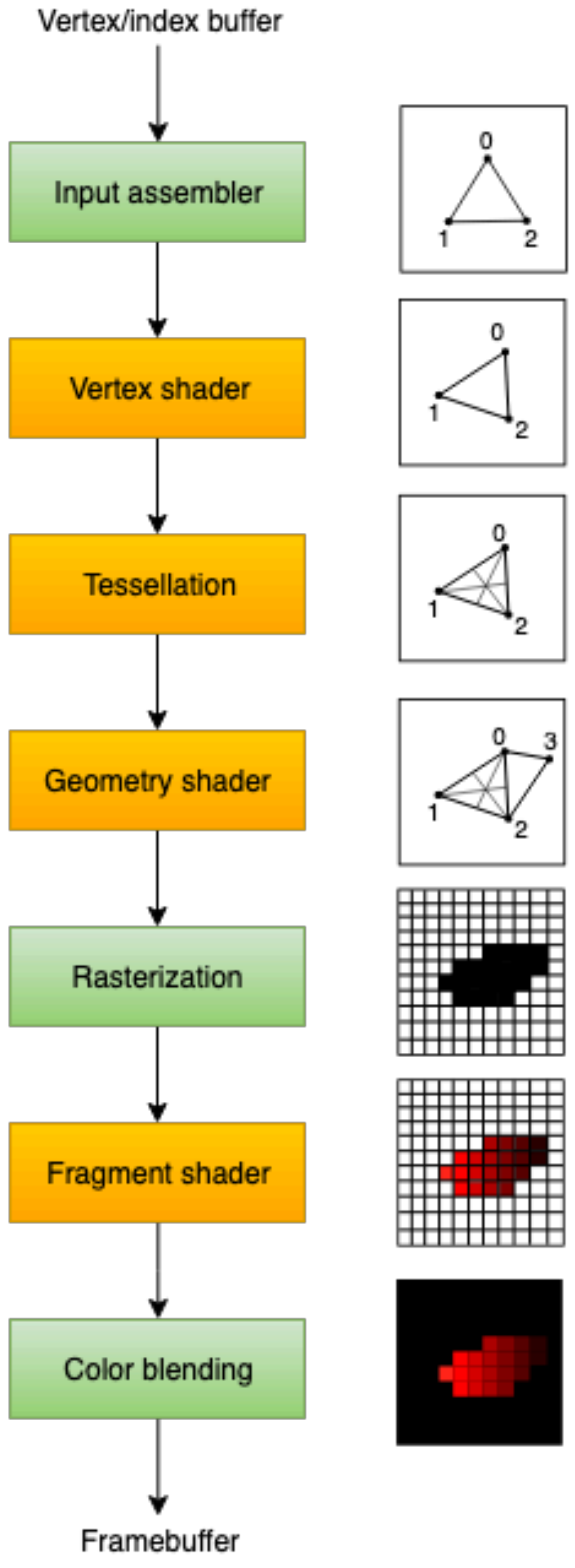


3. Post-processing
Apply additional image effects

* These images are simplified representations. The actual number of pixels is much higher.

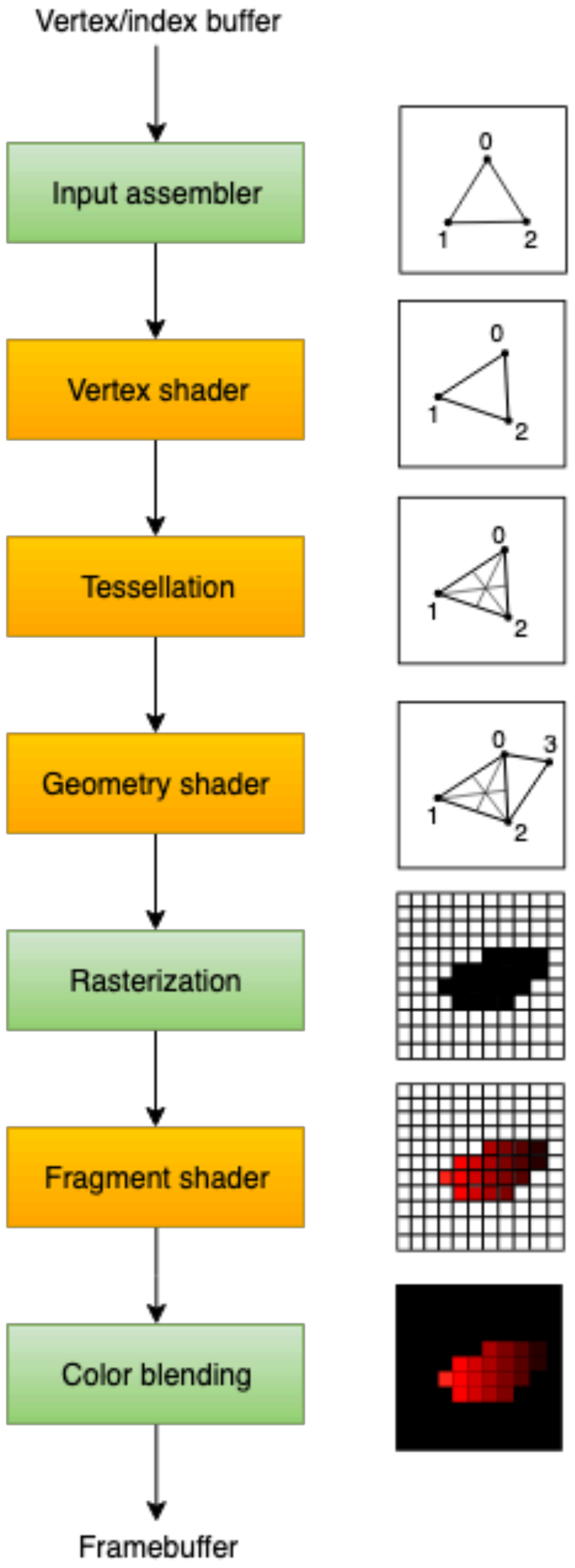
Shading pipeline



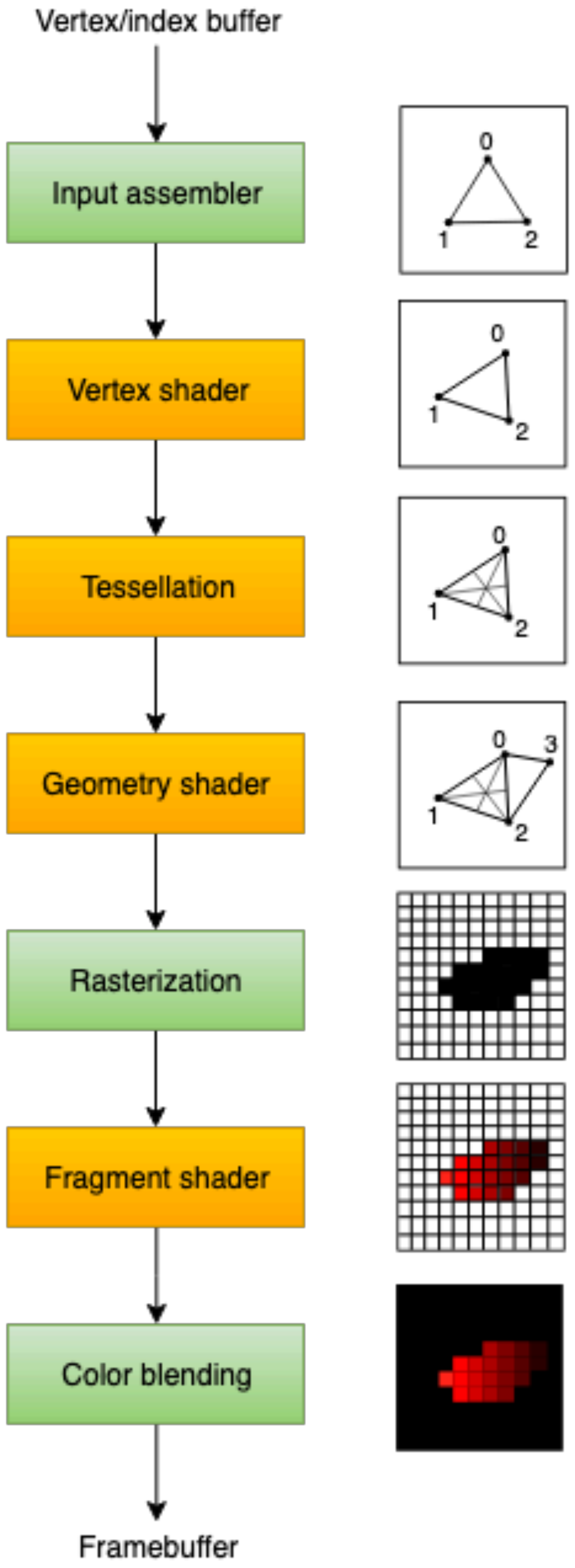


Shader are written in Nvidia CG or HLSL and encapsulated in Unity's own ShaderLab language.

Unity reference
https://docs.unity3d.com/Manual/SL-Shader.html

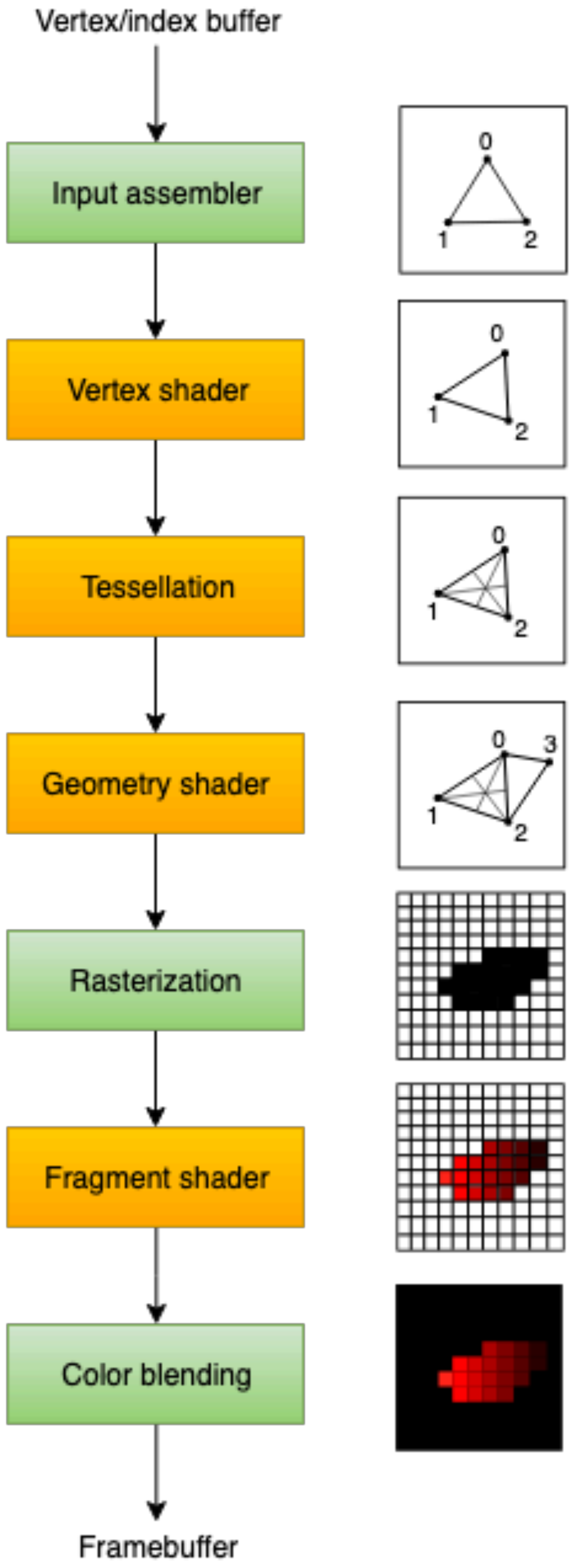



```
Shader "DataVisWorkshop/GLShader"
{
    SubShader
    {
        Tags { "RenderType"="Transparent" "Queue"="Transparent" "IgnoreProjector"="True" }
        Blend SrcAlpha OneMinusSrcAlpha
        ZWrite Off

        Pass
        {
            CGPROGRAM
            #pragma vertex Vert
            #pragma fragment Frag

            #include "UnityCG.cginc"

            struct ToVert
            {
                float4 vertex : POSITION;
                float4 color : COLOR;
            };

            struct ToFrag
            {
                float4 vertex : SV_POSITION;
                float4 color : COLOR;
            };

            ToFrag Vert( ToVert v )
            {
                ToFrag o;
                o.vertex = UnityObjectToClipPos( v.vertex );
                o.color = v.color;
                return o;
            }

            fixed4 Frag( ToFrag i ) : SV_Target
            {
                return i.color;
            }

            ENDCG
        }
    }
}
```



ShaderLab language

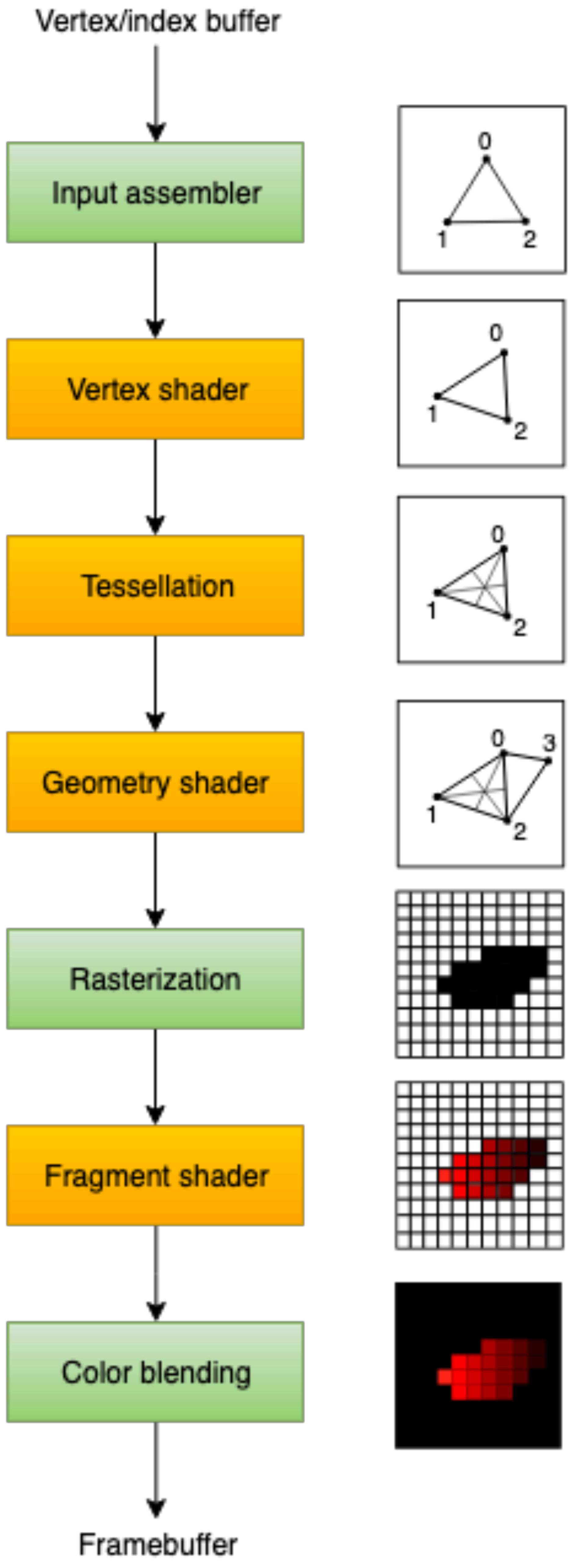
```
Shader "DataVisWorkshop/GLShader"
{
    SubShader
    {
        Tags { "RenderType"="Transparent" "Queue"="Transparent" "IgnoreProjector"="True" }
        Blend SrcAlpha OneMinusSrcAlpha
        ZWrite Off

        Pass
        {
            CGPROGRAM
            #pragma vertex Vert
            #pragma fragment Frag

            #include "UnityCG.cginc"

            struct ToVert
            {
                float4 vertex : POSITION;
                float4 color : COLOR;
            };

            struct ToFrag
            {
                float4 vertex : SV_POSITION;
                float4 color : COLOR;
            };

            ToFrag Vert( ToVert v )
            {
                ToFrag o;
                o.vertex = UnityObjectToClipPos( v.vertex );
                o.color = v.color;
                return o;
            }

            fixed4 Frag( ToFrag i ) : SV_Target
            {
                return i.color;
            }

            ENDCG
        }
    }
}
```



ShaderLab language

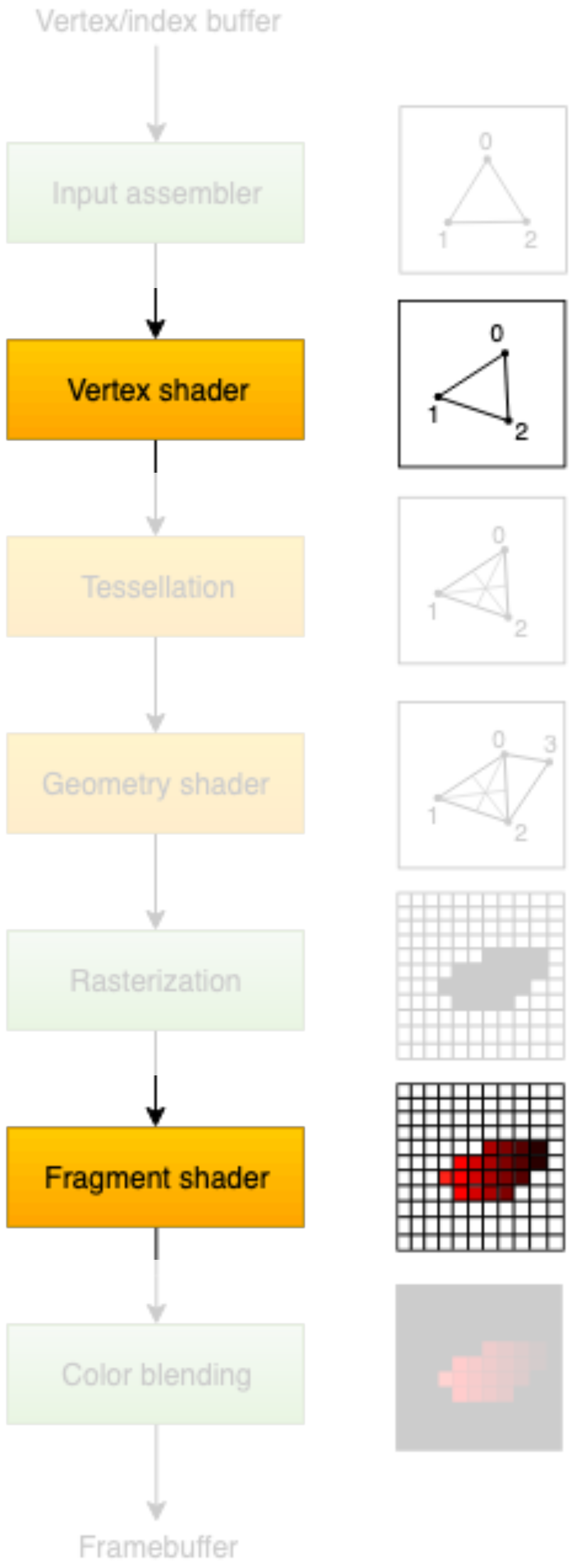
```
Shader "DataVisWorkshop/GLShader"
{
    SubShader
    {
        Tags { "RenderType"="Transparent" "Queue"="Transparent" "IgnoreProjector"="True" }
        Blend SrcAlpha OneMinusSrcAlpha
        ZWrite Off

        Pass
        {
            CGPROGRAM
            #pragma vertex Vert
            #pragma fragment Frag

            #include "UnityCG.cginc"

            struct ToVert
            {
                float4 vertex : POSITION;
                float4 color : COLOR;
            };

            struct ToFrag
            {
                float4 vertex : SV_POSITION;
                float4 color : COLOR;
            };

            ToFrag Vert( ToVert v )
            {
                ToFrag o;
                o.vertex = UnityObjectToClipPos( v.vertex );
                o.color = v.color;
                return o;
            }

            fixed4 Frag( ToFrag i ) : SV_Target
            {
                return i.color;
            }

            ENDCG
        }
    }
}
```

Pass at index 0



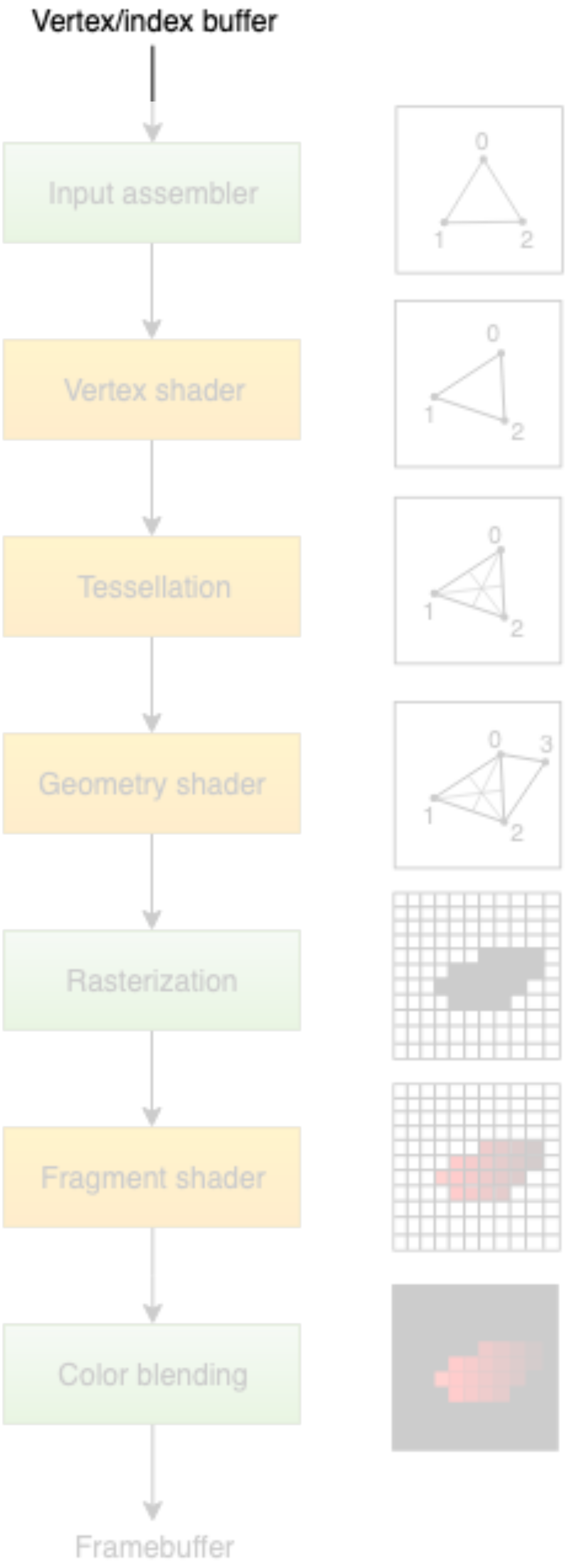
Nvidia CG language

```
Shader "DataVisWorkshop/GLShader"
{
    SubShader
    {
        Tags { "RenderType"="Transparent" "Queue"="Transparent" "IgnoreProjector"="True" }
        Blend SrcAlpha OneMinusSrcAlpha
        ZWrite Off

        Pass
        {
            CGPROGRAM
            #pragma vertex Vert
            #pragma fragment Frag

            #include "UnityCG.cginc"

            struct ToVert
            {
                float4 vertex : POSITION;
                float4 color : COLOR;
            };

            struct ToFrag
            {
                float4 vertex : SV_POSITION;
                float4 color : COLOR;
            };

            ToFrag Vert( ToVert v )
            {
                ToFrag o;
                o.vertex = UnityObjectToClipPos( v.vertex );
                o.color = v.color;
                return o;
            }

            fixed4 Frag( ToFrag i ) : SV_Target
            {
                return i.color;
            }

            ENDCG
        }
    }
}
```



Vertex and fragment function

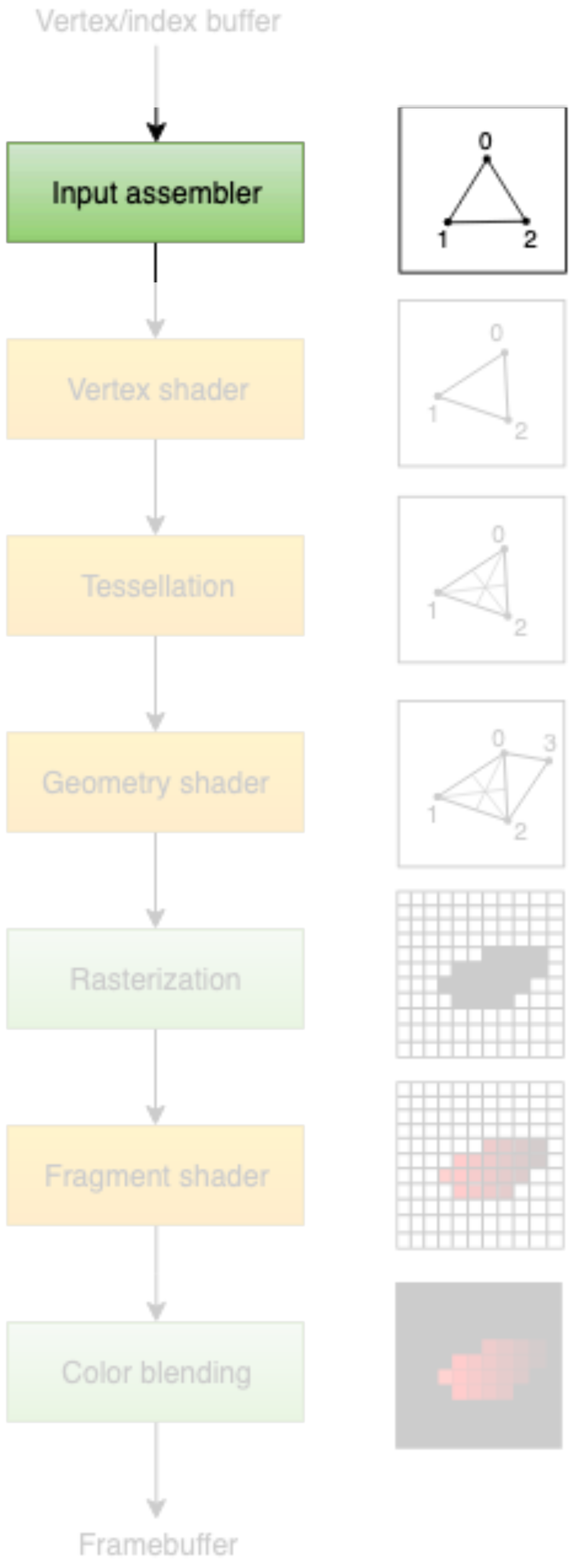
```
Shader "DataVisWorkshop/GLShader"
{
    SubShader
    {
        Tags { "RenderType"="Transparent" "Queue"="Transparent" "IgnoreProjector"="True" }
        Blend SrcAlpha OneMinusSrcAlpha
        ZWrite Off

        Pass
        {
            CGPROGRAM
            #pragma vertex Vert
            #pragma fragment Frag

            #include "UnityCG.cginc"

            struct ToVert
            {
                float4 vertex : POSITION;
                float4 color : COLOR;
            };

            struct ToFrag
            {
                float4 vertex : SV_POSITION;
                float4 color : COLOR;
            };

            ToFrag Vert( ToVert v )
            {
                ToFrag o;
                o.vertex = UnityObjectToClipPos( v.vertex );
                o.color = v.color;
                return o;
            }

            fixed4 Frag( ToFrag i ) : SV_Target
            {
                return i.color;
            }

            ENDCG
        }
    }
}
```



You set a shader and a pass index for that shader. Then you upload index and vertex data in your C# script. For example using the GL class.

This step is expensive (slow) because the CPU has to pass the information to the GPU.

```
material.SetPass( 0 );

// Draw a triangle.
GL.Begin( GL.TRIANGLES );
GL.Vertex3( 0, 0, 0 );
GL.Vertex3( 0, 1, 0 );
GL.Vertex3( 1, 1, 0 );
GL.End();
```

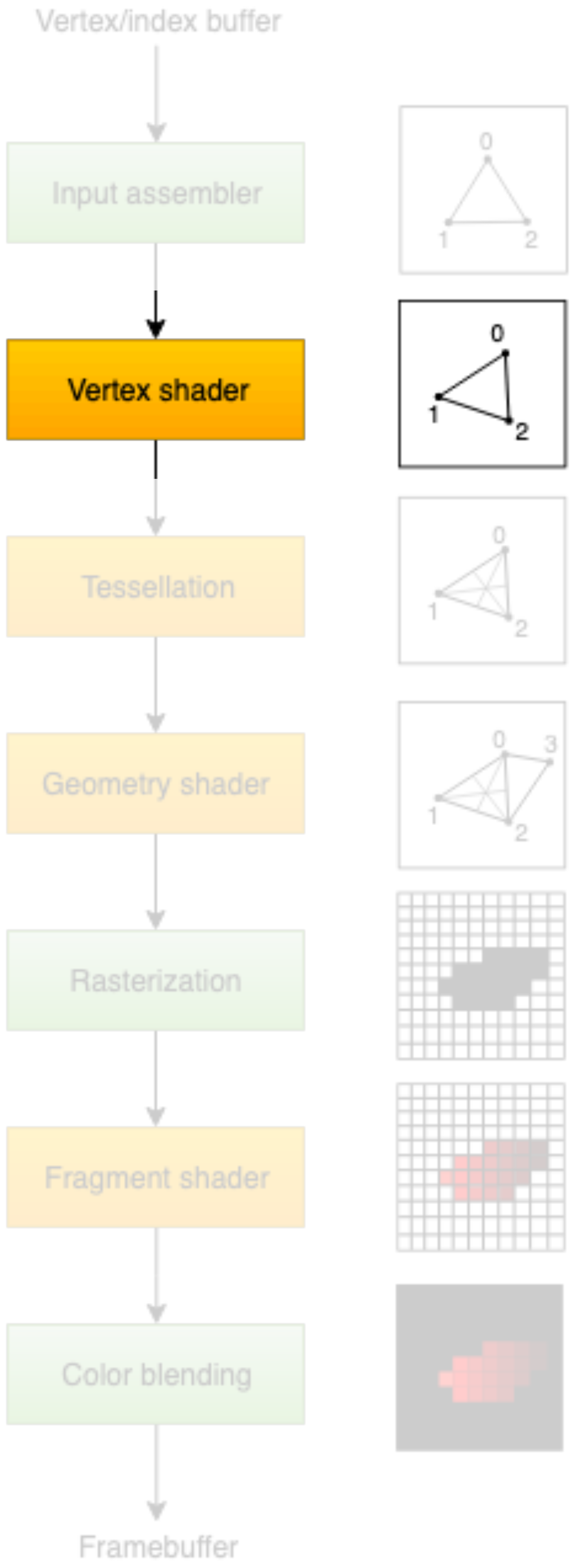


In your shader, you write a struct that informs the GPU what vertex data to assemble and forward to your vertex function.

```
struct ToVert
{
    float4 vertex : POSITION;
    float4 color : COLOR;
};
```

... you also write a struct that informs the GPU what data you promise to return (send forward) from your vertex function.

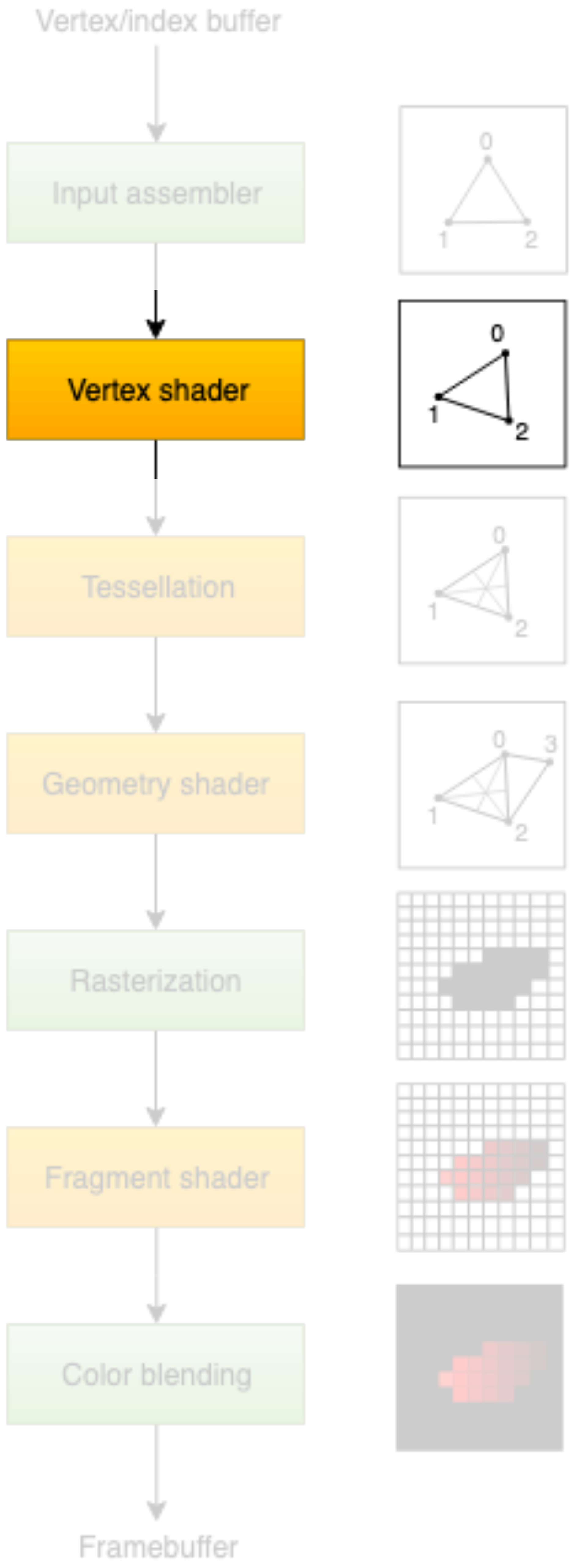
```
struct ToFrag
{
    float4 vertex : SV_POSITION;
    float4 color : COLOR;
};
```



In the vertex function, you transform the vertices for your liking and compute other data to forward.

The vertex function below is called by many GPU thread simultaneously, making these transformations very fast.

```
ToFrag Vert( ToVert v )
{
    ToFrag o;
    o.vertex = UnityObjectToClipPos( v.vertex );
    o.color = v.color;
    return o;
}
```



In the vertex function, you transform the vertices for your liking and compute other data to forward.

The vertex function below is called by many GPU thread simultaneously, making these transformations very fast.

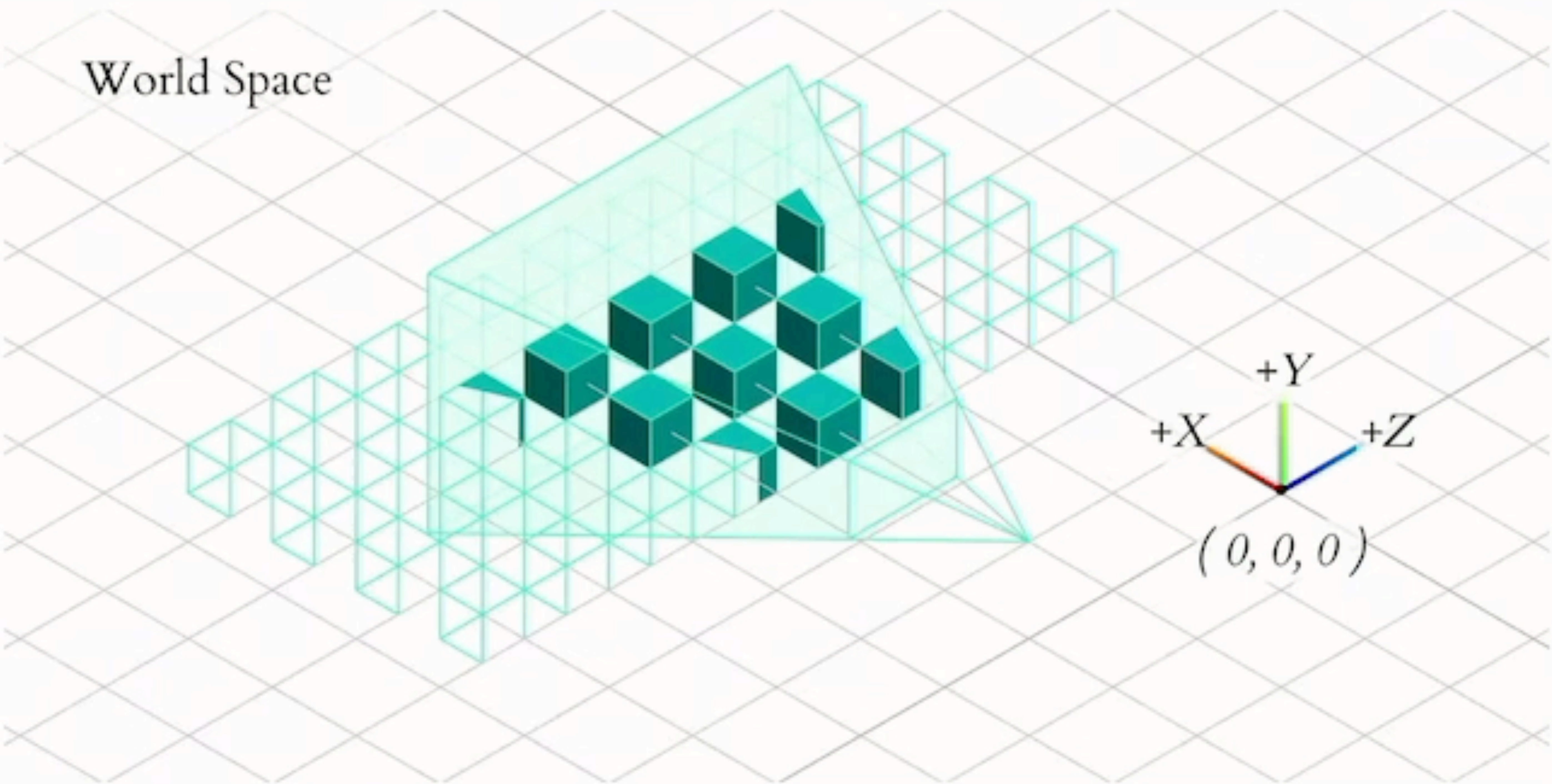
```
ToFrag Vert( ToVert v )
{
    ToFrag o;
    o.vertex = UnityObjectToClipPos( v.vertex );
    o.color = v.color;
    return o;
}
```

model-view-projection transformation.

From model to clip space

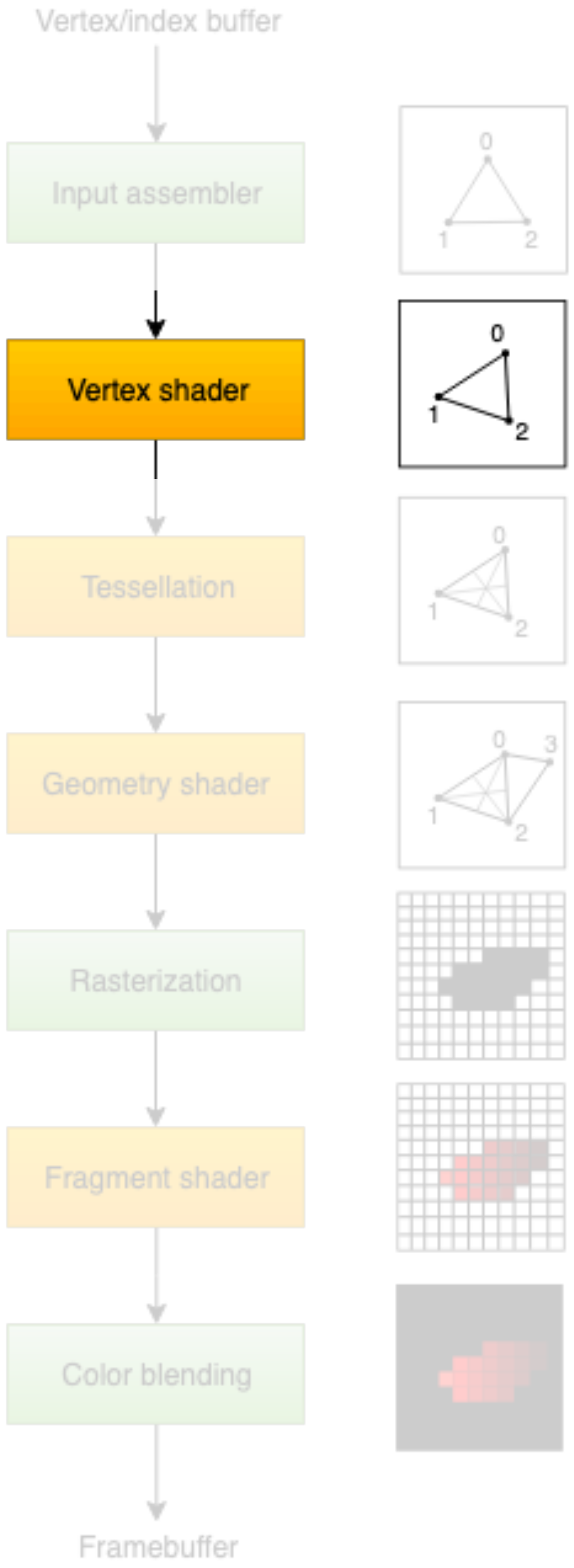
```
GL.MultMatrix( matrix );

Graphics.DrawMesh( _mesh, transform.localToWorldMatrix, material, gameObject.layer );
```

Combined with
camera worldToLocal matrix and
camera projection matrix



A scene being visualized in world space, camera space, and then normalized device coordinates, representing the stages of transformation in the *Model View Projection* pipeline.

https://jsantell.com/model-view-projection/

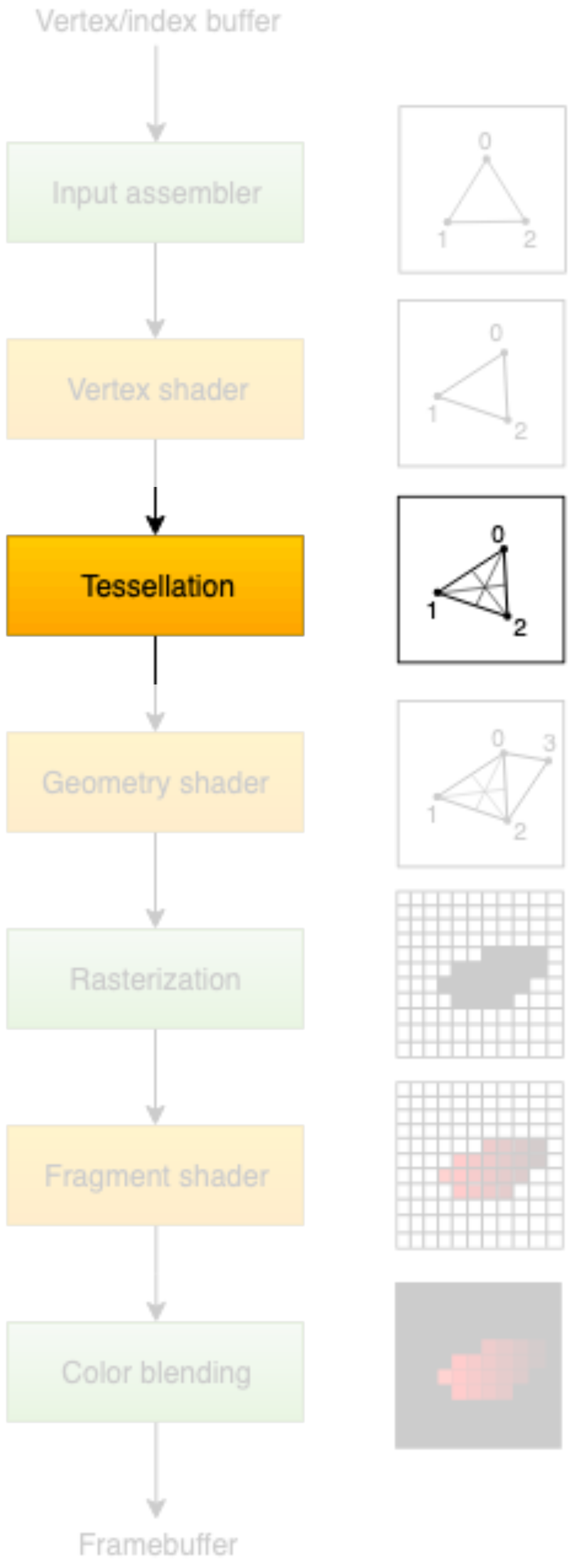


In the vertex function, you transform the vertices for your liking and compute other data to forward.

The vertex function below is called by many GPU thread simultaneously, making these transformations very fast.

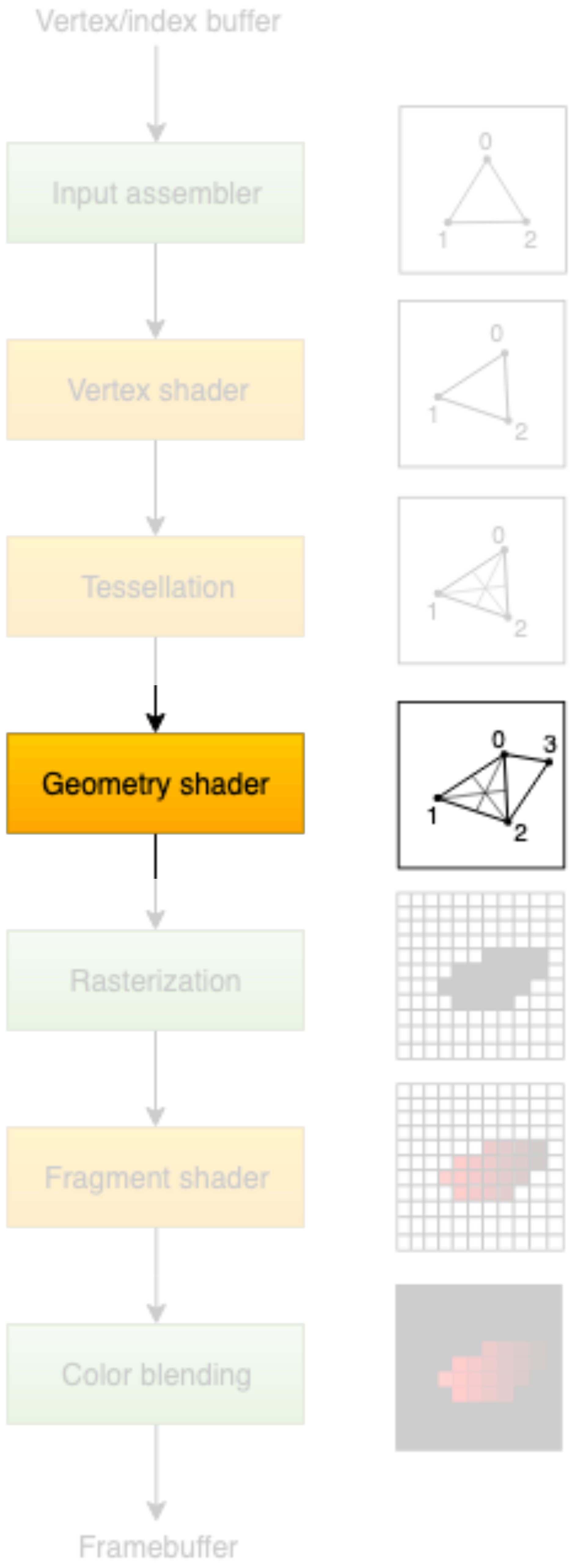
```
ToFrag Vert( ToVert v )
{
    ToFrag o;
    o.vertex = UnityObjectToClipPos( v.vertex );
    o.color = v.color;
    return o;
}
```

After the vert function, the clip space vertex is converted to Normalised Device Coordinate (NDC) space by the GPU. https://answers.unity.com/questions/1443941/shaders-what-is-clip-space.html



You (can) write a rather complex function to subdivide your triangles into smaller parts. This is used to generate beautifully rounded surfaces.

Tutorial here:
https://catlikecoding.com/unity/tutorials/advanced-rendering/tessellation/

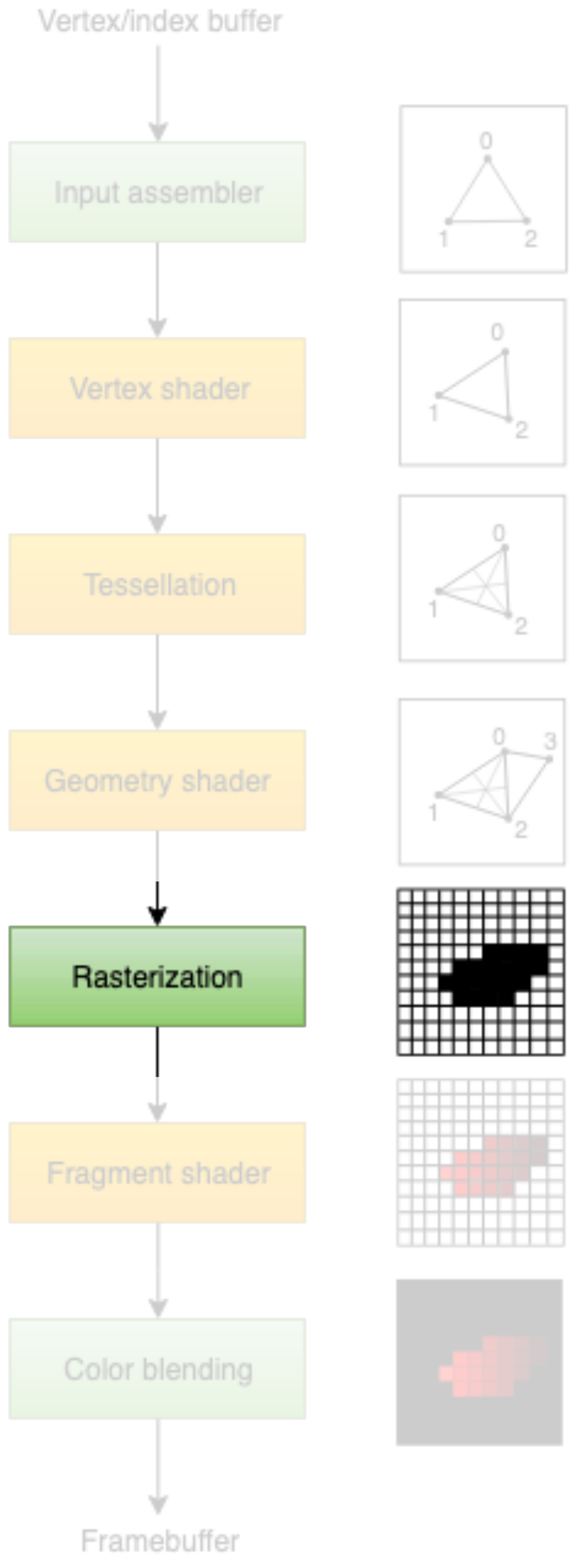


For the platforms that support it, you (can) write a "geometry" function that expand each input vertex. This is for example useful for expanding particle positions into quads, thereby reducing vertex data upload .

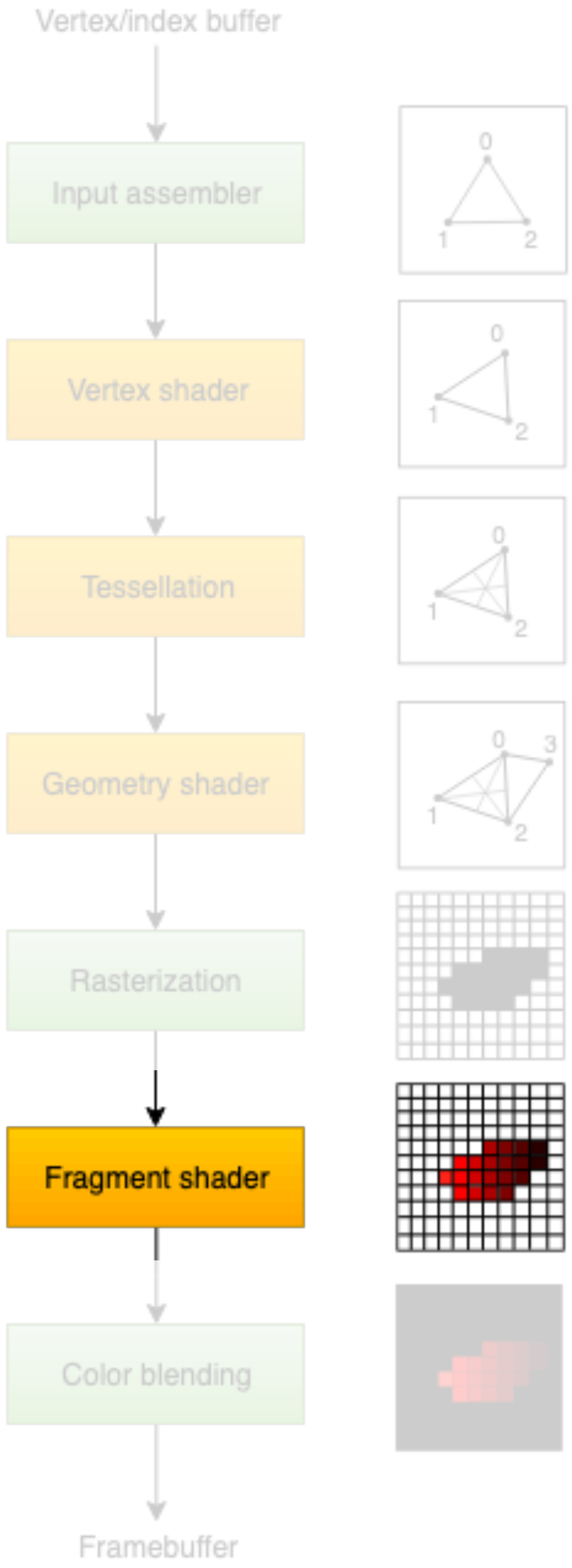
Not supported on macOS, iOS and most Android devices.

Supported by most modern desktop graphics cards.

```
[maxvertexcount(4)]
void Geom( point Empty primitive[1], uint i : SV_PrimitiveID, inout TriangleStream<ToFrag> triStream )
{
    // Read.
    float4 posRadius, color;
    ReadPoint( i, posRadius, color );
    #if defined(_OPTION_B) || defined(_OPTION_C)
        float3 vel = _Velocities[i].xyz;
    #endif

    // Compute expansion.
    #ifdef _OPTION_B // Velocity to orientation
        float3 rightExtents = normalize( cross( CameraForward(), vel ) );
        float3 upExtents = normalize( cross( rightExtents, CameraForward() ) );
    #else
        float3 rightExtents = CameraRight();
        float3 upExtents = CameraUp();
    #endif

    #ifdef _OPTION_C // Velocity stretch
        float speed = length( vel );
        half stretch = 0;
```



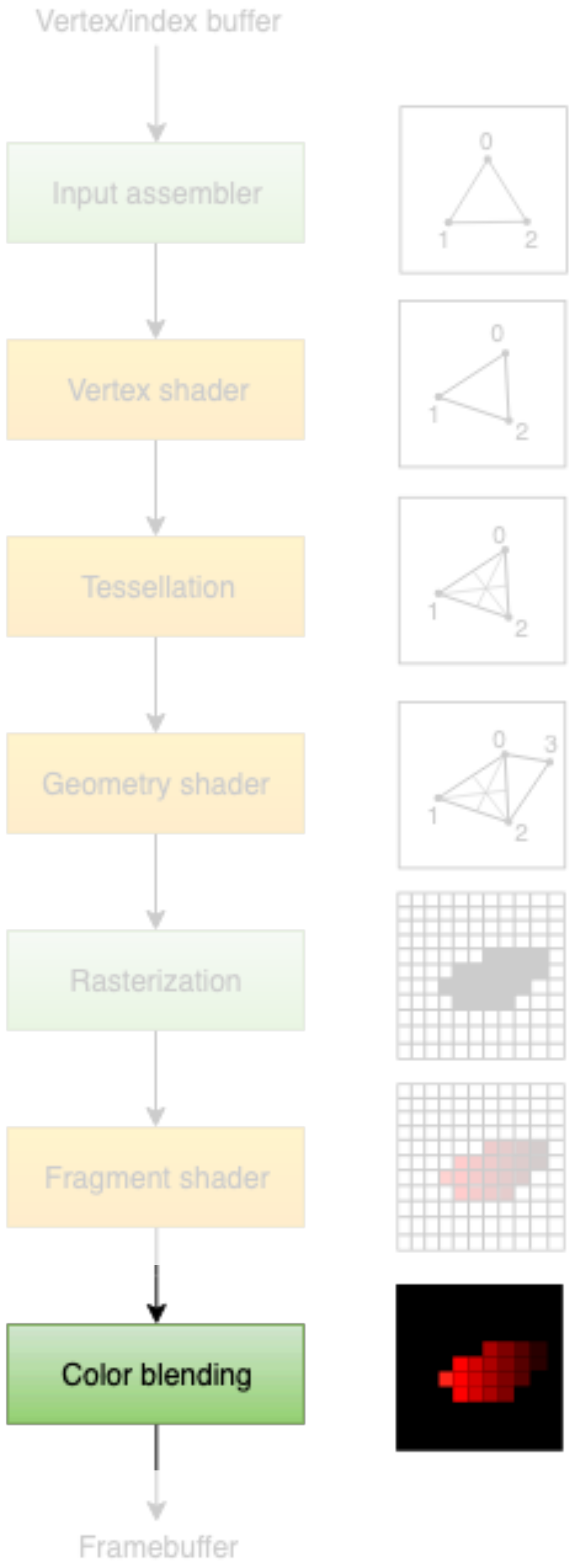
The GPU figures out that parts of your geometry ends up inside the pixels of your camera view. A huge number of GPU threads are started to process as many pixels simultaneously as possible.



In the fragment function, you receive data from the vertex function (or geometry or tessellation functions).

This is where you typically read from textures and compute lighting. The function is also used for fullscreen effects and pixel based simulations.

```
fixed4 Frag( ToFrag i ) : SV_Target
{
    return i.color;
}
```



Depending on the blend mode you have defined in your shader pass, the new color is blended with existing color for that pixel.

Unity reference
https://docs.unity3d.com/Manual/SL-Blend.html

```
Blend SrcAlpha OneMinusSrcAlpha
```