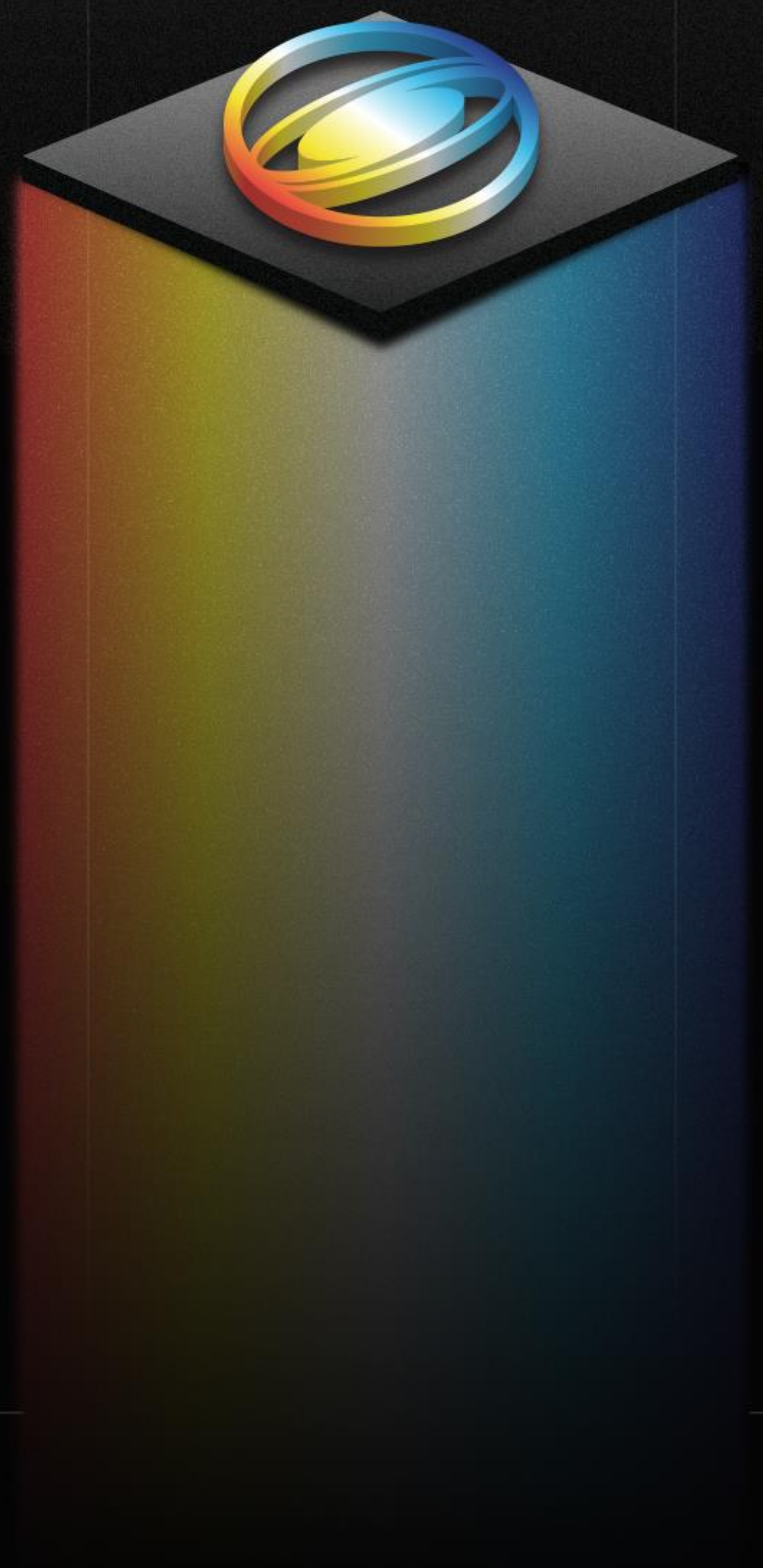
SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

THE PREMIER CONFERENCE
& EXHIBITION ON
COMPUTER GRAPHICS &
INTERACTIVE TECHNIQUES

HYDRA GRAPHICS INTERFACE

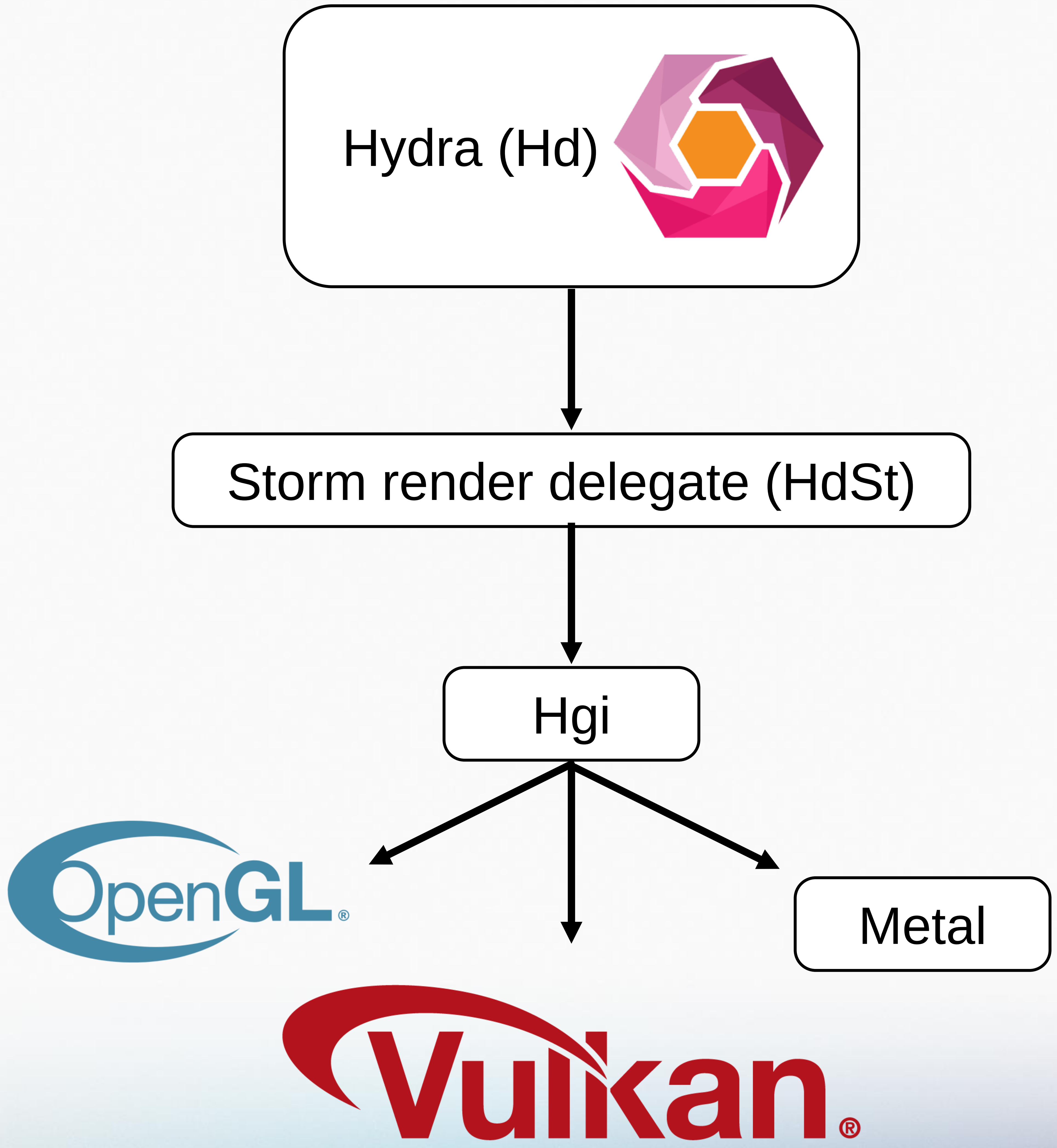


CAROLINE LACHANSKI
PIXAR ANIMATION STUDIOS
CLACHANSKI@PIXAR.COM

TOM CAUCHOIS
PIXAR ANIMATION STUDIOS
TCAUCHOIS@PIXAR.COM

HYDRA GRAPHICS INTERFACE (HGI)

- **Hydra** originally an OpenGL-based renderer
  - Meant as ground truth visualization for USD
- OpenGL render delegate component became **Storm**
  - Used in apps like usdview and Presto
- **Hgi** is graphics API abstraction layer
  - HgiGL currently used internally
  - HgiMetal result of collaboration with Apple
  - HgiVulkan now the focus
- Pixar goal to shift from OpenGL to Vulkan internally
- How to write renderer independent of graphics API without disrupting users?



Hydra (Hd)

Storm render delegate (HdSt)

Hgi

OpenGL

Metal

Vulkan

TRANSITIONING TO HGI

- Storm written with OpenGL in mind, Hgi written with modern APIs in mind
- OpenGL state machine to explicit pipeline
  - HgiVulkan: commands are recorded in command buffer → command buffer is submitted
  - HgiGL: functions are accumulated in stack → GL state captured → functions (GL calls) called → GL state restored
- Lingering GL code and GL concepts
- Vulkan validation layers

```
HgiGLOpsFn
HgiGLOps::SetViewport(GfVec4i const& vp)
{
    return [vp] {
        glViewport(vp[0], vp[1], vp[2], vp[3]);
    };
}
```

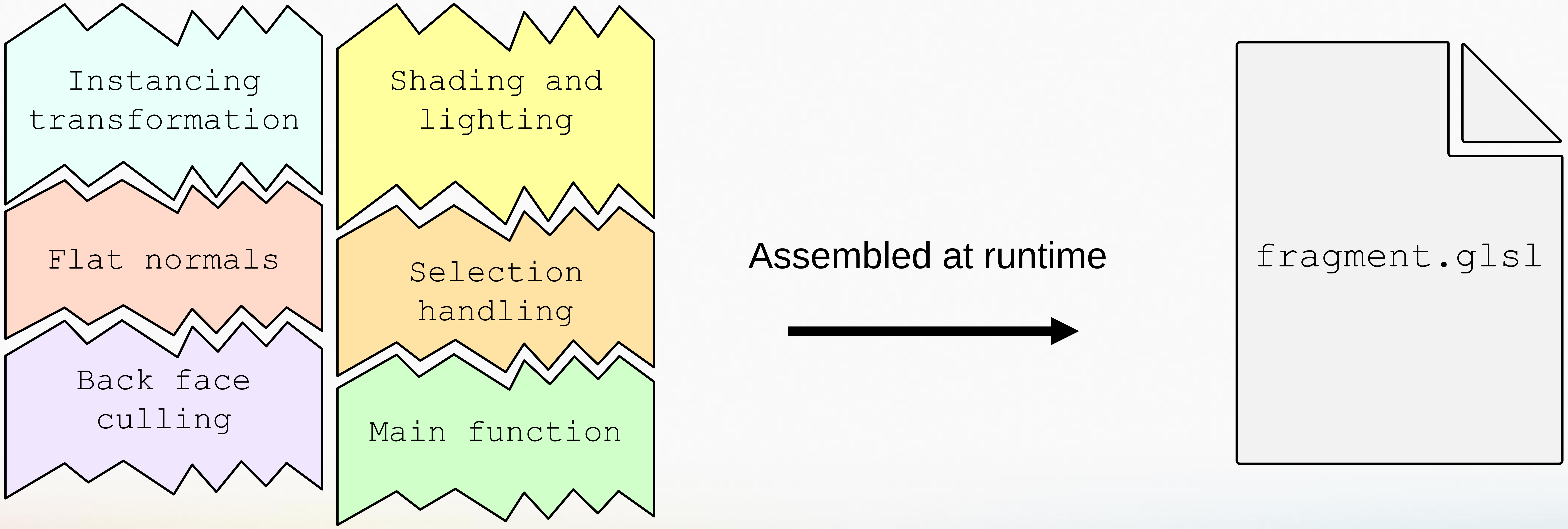
COORDINATE SYSTEM DIFFERENCES

- Had to deal with Vulkan coordinate systems
  - Storm uses OpenGL-style projection matrix, assumes bottom-left origin for viewport
- Originally set negative height for Vulkan viewport
- But with this, also needed to:
  - Negate shader `dFdy` results
  - Change `gl_FragCoord.y` to `(1 - gl_FragCoord.y)`
  - Change how we sampled from AOVs in the shader
  - NOT flip image when writing to disk

- Ended up using OpenGL-style projection matrix with non-negative viewport, but flipping the winding order
  - Resulting image is upside down, which works well in our system
  - Only extra work is to flip the image vertically during interop

GLSLFX

- **GLSLFX** is domain language for defining shader pipelines in Storm
  - Defines imports, configurations, and shading code snippets



Instancing transformation

Flat normals

Back face culling

Shading and lighting

Selection handling

Main function

Assembled at runtime

fragment.glsl

Multiple shader snippets

Completed fragment shader

HOW TO WRITE SHADER RESOURCES?

- GLSL is original shading language of choice
- Shader resources originally hardcoded in shader snippets
  - Shader stage inputs and outputs
  - Texture and data buffer declarations
  - Interpolation modifiers
  - Location and binding indices
  - Other layout qualifiers (e.g. “early_fragment_tests” for the FS)

- Wanted shader language-independent way of declaring shader’s resources and resource layout

SHADER RESOURCE LAYOUTS

- Extended GLSLFX to include “layout” section
- Corresponds to “glsl” section of same name
- Processed at runtime to fill descriptors, which are processed by shadergen to produce shading code

```
--------------------------------------------------------------------
-
-- glsl Mesh.Vertex

out VertexData
{
    vec4 Peye;
    vec3 Neye;
} outData;

void main(void)
{
    outData.Peye = [. . .];
    outData.Neye = [. . .];
    gl_Position = vec4(GetProjectionMatrix() * outData.Peye);
}
```

Before resource layouts

```
--------------------------------------------------------------------
-- layout Mesh.Vertex

[
    ["out block", "VertexData", "outData",
        ["vec4", "Peye"],
        ["vec3", "Neye"]
    ]
]

---------------------------------------------------------------------
-- glsl Mesh.Vertex

void main(void)
{
    outData.Peye = [. . .];
    outData.Neye = [. . .];
    gl_Position = vec4(GetProjectionMatrix() * outData.Peye);
}
```

With resource layouts

SHADER GENERATION

- API-specific shader creation is handled with Hgi shadergen system
- Set of classes that generate API-specific shading code
- Fed by descriptors:
  - `HgiShaderFunctionTextureDesc`, `HgiShaderFunctionBufferDesc`, `HgiShaderFunctionFragmentDesc`, etc.
- Behind abstraction layer, we can deal with resource declaration, builtin function and keyword name differences, extension names, etc.

```
struct HgiShaderFunctionTextureDesc
{
    std::string nameInShader;
    uint32_t dimensions;
    uint32_t bindIndex;
    size_t arraySize;
    bool writable;
    . . .
};
```

SHADER GENERATION EXAMPLE

- OpenGL GLSL builtin vertex stage input variables `gl_VertexID` and `gl_InstanceID`
- Vulkan GLSL extension replaces* those with `gl_VertexIndex` and `gl_InstanceIndex`
- We want shader writers to be able to use these variables without having to think about the backend differences
- Map variables “`hd_VertexID`” and “`hd_InstanceID`” to a non-backend-specific role
- Each backend’s shadergen emits code defining `hd_VertexID and hd_InstanceID` to correct thing

OpenGL GLSL:

```
uint hd_VertexID = gl_VertexId;
uint hd_InstanceID = gl_InstanceId;
```

Vulkan GLSL:

```
uint hd_VertexID = gl_VertexIndex;
uint hd_InstanceID = gl_InstanceIndex;
```

Metal shading language:

```
uint hd_VertexID[[vertex_id]],
uint hd_InstanceID[[instance_id]],
```

HGIVULKAN SCREENSHOTS

P X A R
ANIMATION STUDIOS