GLSL

Dr. Hamish carr & Dr. Rafael Kuffner dos Anjos

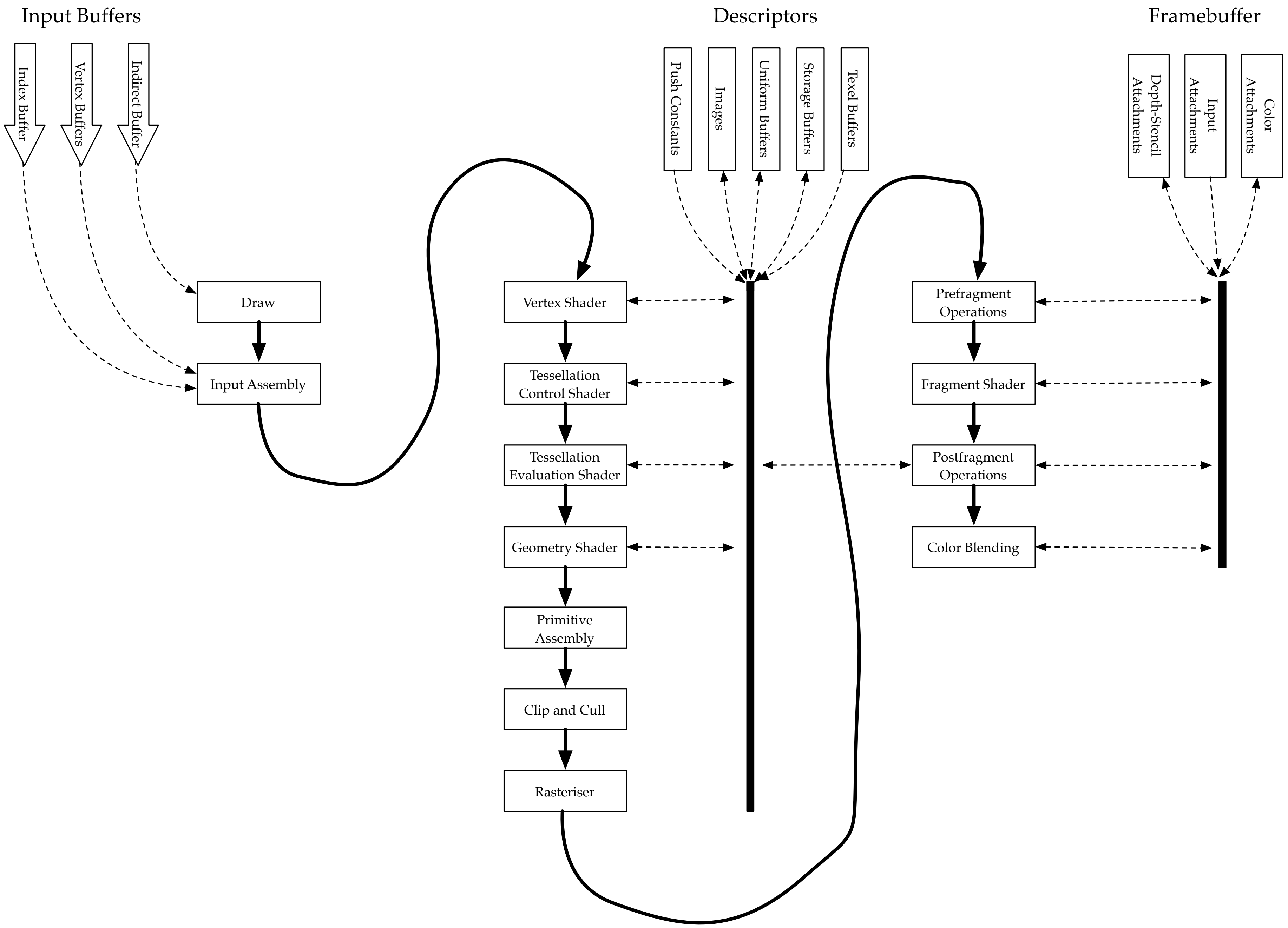
Agenda

- What is a shader
- Shader stages
- Writing a shader

What is a Shader?

- Function called per each element that needs to be processed by the pipeline
  - For each vertex -> Invoke vertex shader
  - For each fragment -> Invoke fragment shader
  - etc
- Essentially, the innards of a loop
- Applied in parallel to many data

The Vulkan Pipeline

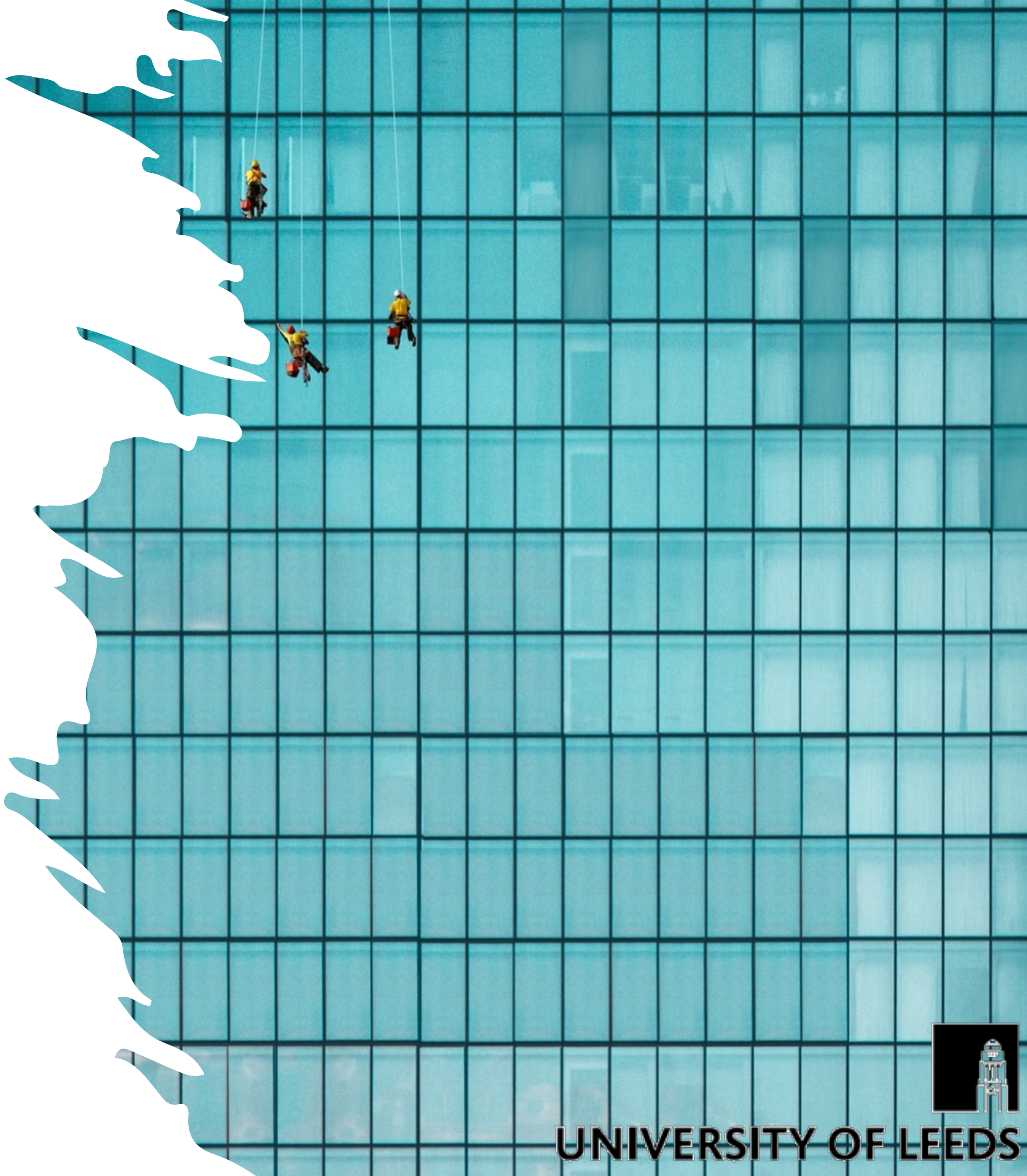


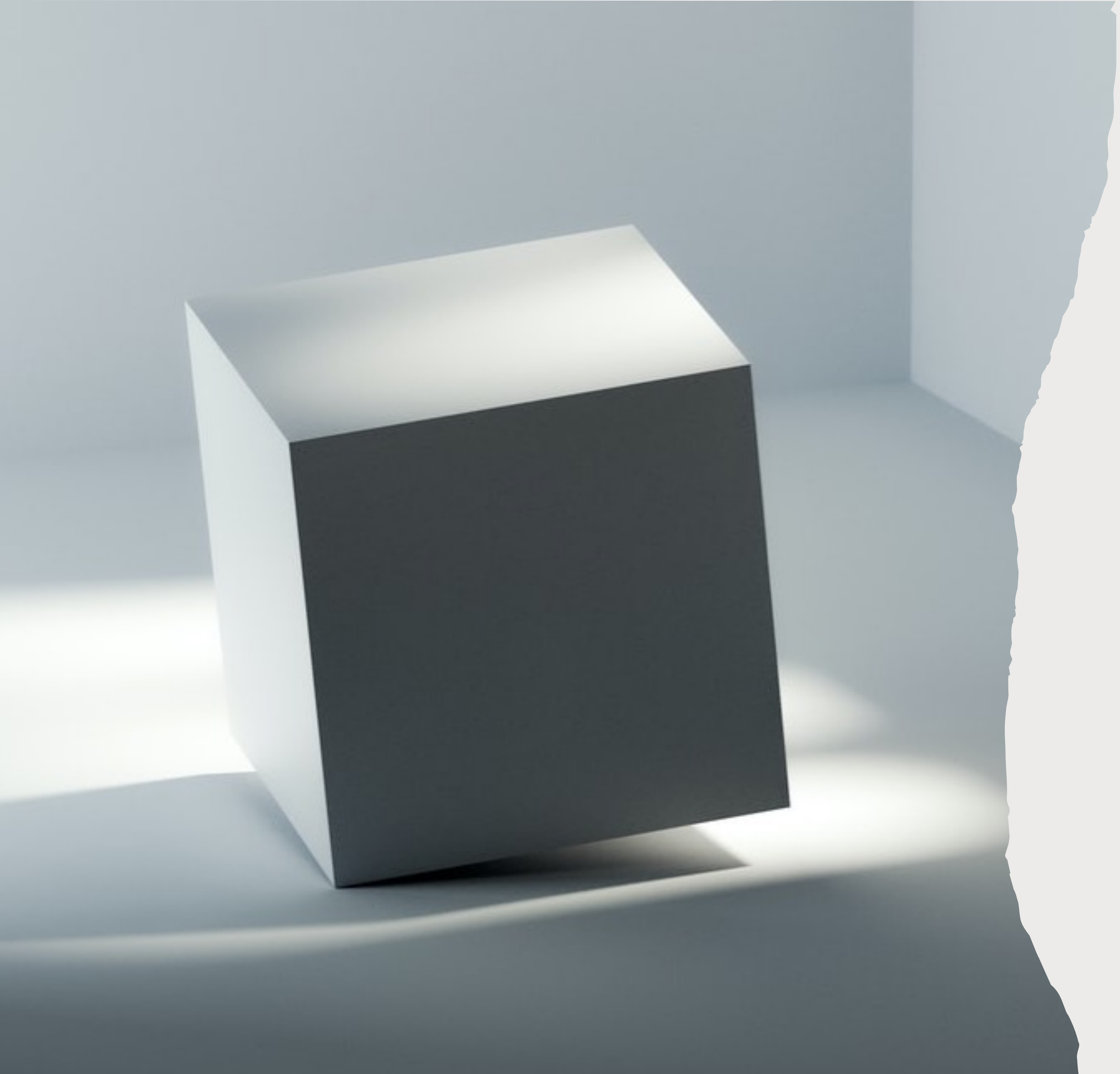
Early Stages

- Draw:
  - Commands enter the pipeline
- Input Assembly:
  - Reads index/vertex buffers
- Vertex Shader:
  - Transforms & processes the vertices

Tessellation Stages

- Tessellation Control Shader:
  - Generates patch tessellation commands
- Tessellation Primitive Generation:
  - Breaks patches into smaller patches
- Tessellation Evaluation Shader:
  - Sets attributes for new vertices
  - Similar to vertex shader





Geometric Stages

- Geometry Shader
  - Operates on full primitives
  - Can change primitive type
- Primitive Assembly
  - Preps vertices for rasterisation
- Clip & Cull
  - Early discard for offscreen primitives

Rasterisation

- Many options, but basically fixed function
- Rasterises & generates fragments
- Computes barycentric coordinates
- Uses them to interpolate attributes

Fragment Stages

- Prefragment Operations
  - Early discard before shading (depth, stencil)
- Fragment Assembly
  - Groups data for fragment shader
- Fragment shader
  - Code for doing shading / rendering

Fragment Processing

- Stages:
  - Scissor test – use a rectangle to clip rendering
  - Depth test – use the z-buffer to discard
  - Stencil test – use a bitmap to clip rendering
  - These can be performed *early* or *late (default)*
  - Relative to the fragment shader

Final Stages

- Postfragment Operations
  - Deferred prefragment operations
  - If Fragment shader changes data
- Color Blending
  - Updates the Framebuffer
  - Performs image processing

Writing Shaders

A shader is therefore just a small program with a main() routine, known as an entry point

We need to discuss:

- Compilation
- Language
- I/O

Shader Compilation

Can be done:

- Runtime (GLSL)
- Precomputed (CUDA)
- Intermediate Code (SPIR-V)

All of which have advantages / disadvantages

SPIR-V Compilation

Shaders precompiled to modules (libraries)

Collections of functions with entry points

Each has a name

And a type (which pipeline stage it is for)

Stored as a stream of 4B words

Essentially, an opcode / bytecode like Java

Can be inspected with spirv-dis

GLSL

- Essentially a dialect of C
- With some C++ conveniences
- We (like Vulkan) will use GLSL for this and next lecture
- Others are similar
- Most of the standard library routines built-in
  - Except memory allocation & I/O

GLSL Types

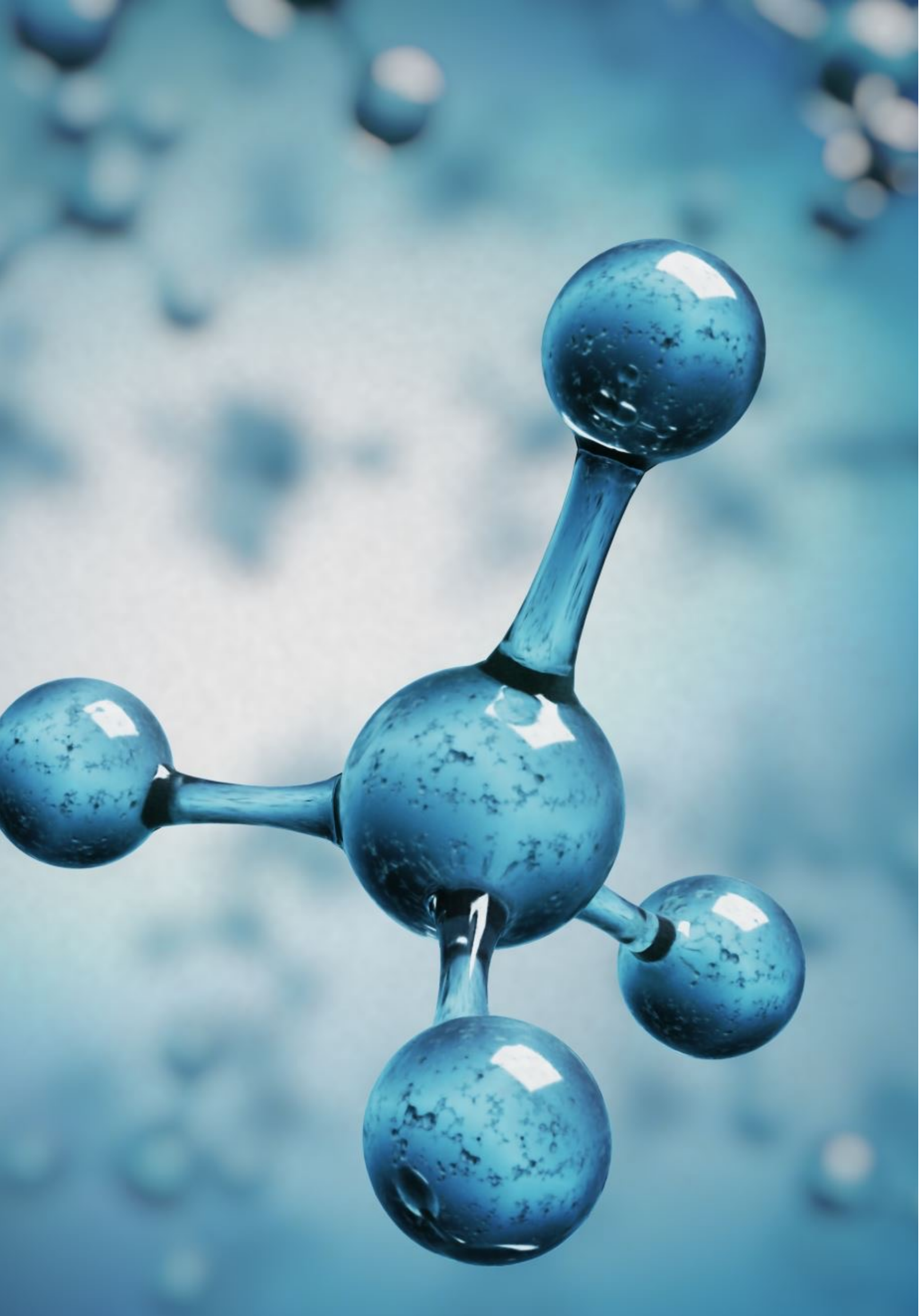
- `bool`: boolean type, as C++
- `int`/ `uint`: basic integer type (usually 4B)
- `float` / `double`: IEEE floating point
  - float is often a lot faster than double
- `vec2`, `vec3`, `vec4`: floating point vectors
  - integers / doubles also available
- `mat2`, `mat3`, `mat4`: matrices

Matrix Types

- Float & double are available, no int or bool
- Represented in column-major format
- Can use m[3] to refer to column 3
- All usual arithmetic operations defined
- But you'll still need a library on CPU

I/O

- Shaders have no print routines
  - Makes it harder to debug
- Instead, they have shared buffers
  - Which change from time to time
  - So they need to be declared explicitly
- Based on *resources* and *descriptors*
  - Resource: a variable outside the shader
  - Descriptor: a bundle of resources



Descriptor Set

- Set of resources bound as a group
- Typically textures, samplers or buffers
- Set up as part of the pipeline
- Then bound to the inputs of the shader
- Push constants are a special case
  - Variables set *directly* from command buffer
  - All others are set by storage in a buffer

Shader Memory Access

Shaders can have local variables

These are usually in registers

Anything else is a resource (three kinds):

- Uniform Blocks (Read Only)
- Texel Buffers (Read Only)
- Shader Storage Blocks (Read/Write)

Read Only solves parallel problems

Uniform Blocks

- Read-only memory
- Shared between all invocations of the shader
- Limited size
- Usually the fastest memory (ie. cache)
- Declared with the uniform keyword

UNIVERSITY OF LEEDS

Uniform Buffers

- Vertex (shader storage) buffers hold attributes
  - Values that change for every vertex
  - Expensive to change (20K vertices => 2MB)
  - Should be loaded at startup/level load
- Uniform buffers are for constants
  - These are the control variables
  - Often small in size (< 1KB)
  - Can be modified every frame

UNIVERSITY OF LEEDS

Types of Uniform Data

Transformation matrices

- Small, change every frame

Flags & constants to control rendering

- Small, may change every frame

Textures

- Large (MB+), often constant
- Unless they are *generated* in multipass

Transformation Matrices

- We may want:
  - Model matrix
  - View matrix
  - Projection matrix
  - Texture matrix
  - Shadow matrix
  - And many others . . .

Multiple Matrices

- OpenGL had matrix *stacks*
- Useful for animation
- But this broke down with skinning
  - Vertices affected by *multiple* transformations
  - E.g. armpit – skin affected by chest & arm
  - Need to compute an average position
  - So need access to more than one matrix

Vulkan Solution

- Uniform buffer is untyped (like all buffers)
- Store anything you want
- You have to specify alignment / offsets
  - So shader can find the data it needs
- But you can pass anything you want
  - Including 50 different matrices

Uniform Buffer Stages

- Control code specifies a descriptor layout
  - As part of pipeline creation
  - Then allocates a descriptor set from a pool
  - And binds the descriptor set for rendering
- Layout (again) tells shader how to access it
- Very similar to vertex buffer setup
  - But don't use a staging buffer (why not?)

Image Buffers (Textures)

Textures are basically constants

- Shared between many vertices
- So they are passed as uniform buffers
- Ideally, no texture swapping during the pass

But they need an extra layer

- A *sampler*
- Which takes care of interpolation / filtering

Texel Buffers

- Read-only
- Can convert formats for you
- Can do interpolation
- Best choice for large arrays of data
- Accessed with texelFetch() function
- Declared as uniform samplerBuffer

Shader Storage Blocks

- Read-write access
- Support atomic operations
- Therefore often slower
- But much larger in size
- Declared with buffer keyword
- For example, a vertex buffer
- Or a frame buffer

A Simple Vertex Shader

```
// Taken from the Vulkan Tutorial
// These are needed to specify version, &c. to the compiler
#version 450
#extension GL_ARB_separate_shader_objects: enable

/////////////////////////////////////////////////////////////
//                                                         //
// I/O VARIABLES: Set up through pipeline instantiation    //
//                                                         //
/////////////////////////////////////////////////////////////

// This declares an output variable
// I'll leave it to you to work out exactly what the type information means
layout(location = 0) vec3 fragColor;

// Why haven't we declared an input variable?

/////////////////////////////////////////////////////////////
//                                                         //
// LOCAL VARIABLES: instantiated for every invocation      //
//                                                         //
/////////////////////////////////////////////////////////////

// Declare three vertex positions in 2D
vec2 positions[3] =
// And instantiate them
    vec2[]  (
// Notice the explicit invocation of constructors
            vec2(0.0, -0.5),
            vec2(0.5, 0.5),
            vec2(-0.5, 0.5)
            );

 // and the main routine
 void main()
     { // main()
     // gl_Position is one of the default output variables
     // gl_VertexIndex is one of the default input variables
     gl_Position = vec4(positions[gl_VertexIndex], 0.0, 1.0);
     // fragColor we declared ourselves, but it's also output
     fragColor = colors[gl_VertexIndex];
     } // main()
```

And a Fragment Shader

```
// Again, this is needed for the compiler
#version 450
#extension GL_ARB_separate_shader_objects: enable

// the fragment colour (input)
layout(location = 0) vec3 fragColor;

// the output colour
layout(location = 0) vec3 outColor;

// and the main routine
void main()
    { // main()
    // just copy the input colour to the output
    // note that interpolation has already happened
    // in the rasteriser
    // also notice the 1.0 set for the opacity
    outColor = vec4(fragColor, 1.0);
    } // main()
```

Images by

- S5 Ben Lambert
- S6 Victor
- S7 Fakurian Design
- S8 Pascal Bernardon
- S9 Adrianna Geo
- S10Remy Gieling
- S11 Hello I’m Nik
- S21 Ryan Quintal
- S22 Greg Rosenke