Descriptor Sets and Push Constants

Descriptor Sets, reuse with Dynamics, and small values through Push Constants

Uniform Data

- So far, all data passed to shaders has been per-vertex
- What about per-model data? e.g. Projection/View/Model matrices, Lighting data, etc.
  - Referred to as uniform data in OpenGL
- Multiple ways to do this!
- However, all rely on the “PipelineLayout” concept from when Pipeline was created
- Start with “Descriptor Set”...

Descriptor Sets

- What is a Descriptor Set?
  - A set (group) of descriptors!
- So what is a descriptor?
  - A piece of data shared across a draw operation.
- So we can use a single Descriptor Set to describe multiple values being passed into a Pipeline.
- Multiple types of Descriptor Set: Images, Samplers, or in our case a basic “Uniform” Descriptor Set
- Uniform is just a basic resource that can be read from.

Descriptor Set Layout

- Before creating Descriptor Set, need a “Layout” for it to describe how set connects up to a pipeline (i.e. which stage of pipeline and what variable to bind resource to).
- Every set needs a layout
- Pipeline can take multiple sets, and therefore can take multiple layouts
- Layouts are passed into creation of “VkPipelineLayout” when creating Graphics Pipeline

Descriptor Pool

- Still can’t create Descriptor Set! Sets are *allocated* from a pool (similar to Command Buffers from Command Pools)
- Creation of pool needs to know size of pool
  - **VkDescriptorPoolSize**: Used to describe size of pool for one descriptor type (size of ALL descriptors of that type that will be created for it, not just one)
- Pool needs to know maximum number of sets that will be created from it…
- ...but that’s all! Pool creation is simple!

Uniform Buffer

- Before we can create a Descriptor Set, we need a buffer for it to reference. Descriptor data needs to exist somewhere!
- Can reuse buffer creation code from Vertex Buffer creation. Principle is the same.
- If data will change frequently (such as in a model matrix) then should be Host Visible. Creating and destroyed staging buffers every time data updates creates lots of overhead!
- If data is largely static (such as in a texture) then use staging buffers. We will cover this in a later lesson!
- Following Buffer creation, simply vkMapMemory the chosen data to the buffer, as done previously.

Setting Up and Using Descriptor Set

- First need to *allocate* Descriptor Set from Descriptor Pool.
- Then bind Descriptor Set to Buffer. Requires two structs.
  - **VkWriteDescriptorSet**: Describes which part of a Descriptor Set to bind (i.e. Set, Binding, Array Element if value is an array)
  - **VkDescriptorBufferInfo**: Passed into VkWriteDescriptorSet, describes which Buffer to bind to the Descriptor
- vkUpdateDescriptorSets then used to execute binding.
- To use Descriptor Set, bind in Command Buffer
  - **vkCmdBindDescriptorSets**: Describes Descriptor Sets to bind to pipeline. After binding, shaders will have access to bound Descriptor Sets.

Descriptor Sets in Shader

- Important to note, unlike OpenGL, Vulkan uniforms can only be passed across as “Uniform Buffer Objects”, not individual values.

- layout(binding = 0) uniform exampleObject {
      mat4 exampleValue
  } exampleName;

- Similar to C struct, values must be inside the struct.
- Use keyword “uniform”, similar to OpenGL, to define uniform value.
- Layout “binding” must match that of the binding in VkWriteDescriptorSet
- Also important to note: Implicit “set = 0” in layout description.
- Could be written as: layout(set = 0, binding = 0)
- Used for distinguishing between multiple sets. Sometimes useful to put “set = 0” to make code easier to read.

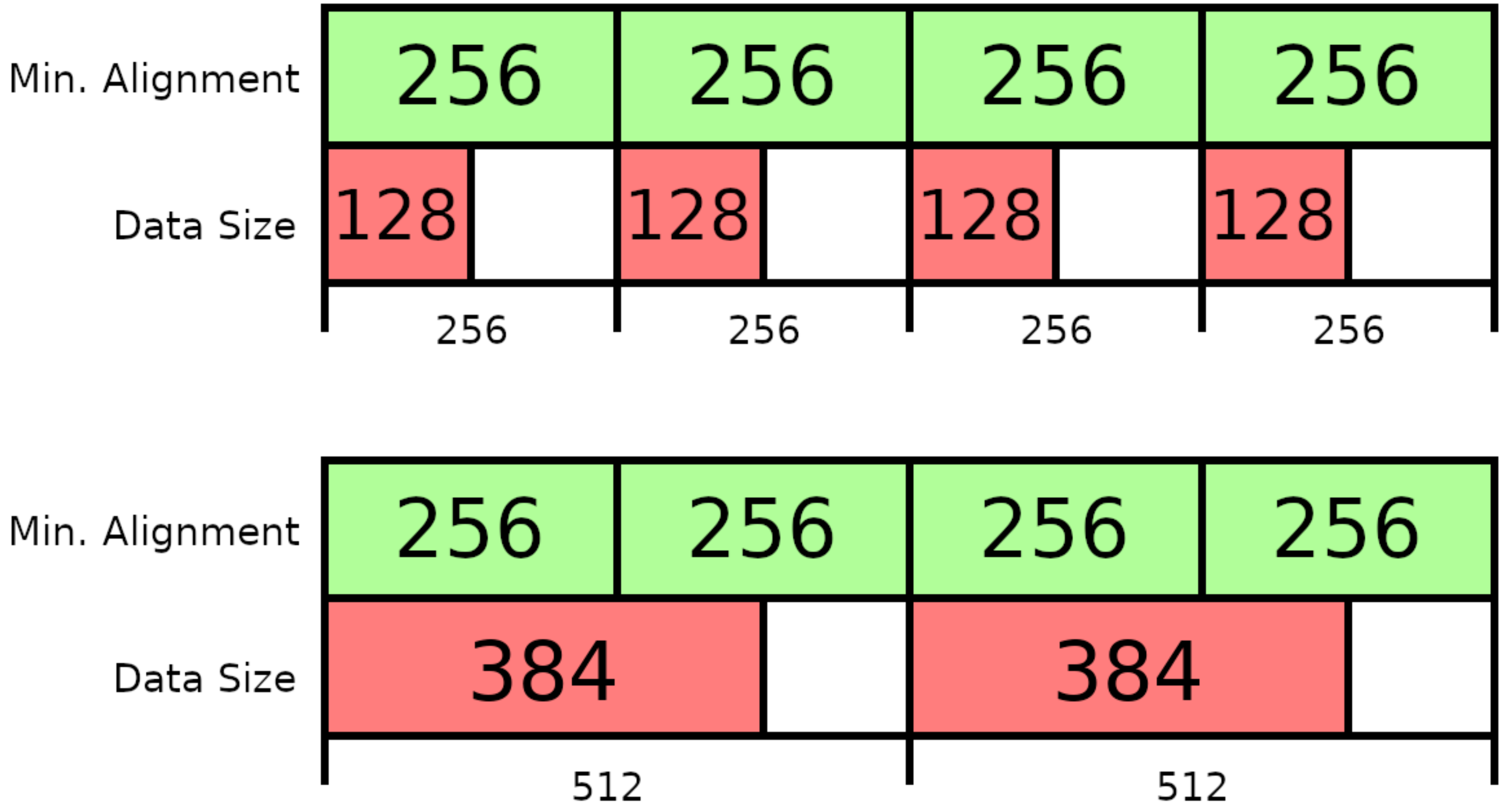
Dynamic Uniforms

- Issue: Uniform Buffers are limited in count. Can only create a small amount before Vulkan restricts creation!
- Need a way to handle hundreds, or thousands of Uniforms.
- Solution: Dynamic Uniform Buffers.
- Similar to Uniform Buffer, but allow holding of multiple Uniform Buffers in large block of memory… and then iterating over data with each draw!
- Means only one Dynamic Uniform Buffer needed for multiple objects.

Dynamic Uniforms (cont.)

- Important: Dynamic Uniform iteration works using minimum data offset. Therefore, need to align data to this value.
- Explained more in coding video…
- … but basically, use bitwise operations to find if our data is smaller than the alignment.
  - If yes, use the minimum alignment.
  - If no, then use the minimum alignment multiplied by how many blocks of minimum alignment the data covers.

Dynamic Uniforms (cont.)



Dynamic Uniforms (cont.)

- Creation of the Dynamic Uniform Buffer is largely the same as a regular Uniform Buffer…
- … But memory must be created with this new alignment instead. This can be done using C function _aligned_malloc.
- When copying data to buffer, it will need to be done by offsetting memory addresses with the previously calculated alignment.
- Additionally, buffer type will now be: VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC
- Lastly, binding the Dynamic Uniform Buffer requires usage of the last two arguments in vkCmdBindDescriptorSets which describe the offsets of each uniform value in the buffer.

Push Constants

- Another issue: Dynamic Uniform Buffers are good for lots of differing data across multiple draws… but if that data is constantly changing, then we have to do a lot of memory operations to update the data!
- Data such as model matrices change very often. We can’t rely on memory operations every frame.
- Solution: Push Constants!
- Push Constants are values that are pushed to the pipeline directly by the command buffer.

Push Constants

- The Good: These are quick pushes that don’t require any buffers or memory operations. The values are passed directly to the pipeline.
- The Bad: Push Constants are limited in size, usually to 128 bytes.
  - Fortunately, things such as model matrices are much smaller than 128 bytes.
  - However, each shader can only have a single Push Constant, limiting us even more.
- The Ugly: We will need to re-record the command buffer every frame!
  - Command Buffer records are very quick though, and don’t add much overhead.

Push Constants (cont.)

- Push Constants are also far easier to set up too!
- First create a “Push Constant Range” to describe the data layout (this is the Push Constant equivalent of a Descriptor Layout).
- Pass the Push Constant Range into the Pipeline Layout.
- And then in the Command Buffer, pass the desired values to the pipeline with vkCmdPushConstants
  - No need for a buffer, you can pass values straight across!
- To reference in a shader, simply change the layout to: layout(push_constant)
- And that’s it!

Uniform vs. Dynamic Uniform vs. Push Constant

- **Uniform Buffers:** Used for unchanging (or rarely changing) values, that are the same across all draw operations.
- **Dynamic Uniform Buffers:** Used for unchanging (or rarely changing) values, that differ across each draw operations.
- **Push Constants:** Used for frequently changing values, that are the same or differ across each draw operation.

Summary

- Descriptor Sets are sets of data that are unchanging across a single draw call.
- We create them by:
  - Defining the layout in a Descriptor Layout
  - Creating a Descriptor Pool
  - Creating a Buffer to hold the descriptor data
  - Allocating a Descriptor Set and binding its descriptors to the appropriate buffers
  - Binding the Descriptor Set in the Command Buffer
- If Descriptor Set data changes between each draw call, a Dynamic Uniform Buffer may be preferable
- If Descriptor Set data changes between each frame, a Push Constant may be preferable

See you next video!