In an effort to expand flexibility and retain efficiency, Vulkan provides something called Push Constants. Like the name implies, these let you “push” constant values out to the shaders. These are typically used for small, frequently-updated data values. This is good, since Vulkan, at times, makes it cumbersome to send changes to the graphics.

By “small”, Vulkan specifies that these must be at least 128 bytes in size, although they can be larger. For example, the maximum size is 256 bytes on the NVIDIA 1080ti. (You can query this limit by looking at the maxPushConstantSize parameter in the VkPhysicalDeviceLimits structure.) Unlike uniform buffers and vertex buffers, these are not backed by memory. They are actually part of the Vulkan pipeline:

```c
VkPushConstantRange vpcr[1];

vpcr[0].offset = 0;
vpcr[0].size = sizeof(glm::mat4);
```

On the application side, push constants are pushed at the shaders by binding them to the Vulkan Command Buffer:

```c
vkCmdPushConstants( CommandBuffer, PipelineLayout, stageFlags,
offset, size, pValues );
```

where:
- stageFlags are or'd bits of VK_PIPELINE_STAGE_VERTEX_SHADER_BIT, VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, etc.
- size is in bytes
- pValues is a void * pointer to the data, which in this 4x4 matrix example, would be of type glm::mat4.

On the shader side, if, for example, you are sending a 4x4 matrix, the use of push constants in the shader looks like this:

```c
layout( push_constant ) uniform matrix
  mat4 modelMatrix;

float     armScale;     // scale factor in x
glm::vec3 armColor;    // color
glm::mat4 armMatrix;   // 4x4 matrix representing arm

struct armArm1
{
    float     armScale;
    glm::vec3 armColor;
    glm::mat4 armMatrix;
};
```

Creating a Pipeline

Prior to that, however, the pipeline layout needs to be told about the Push Constants:

```c
VkPipelineLayoutCreateInfo vplci;

vplci.pSetLayouts = DescriptorSetLayouts;
vplci.setLayoutCount = 4;
```

An example of a pipeline layout that uses push constants:

```c
struct VkPipelineLayoutCreateInfo
{
    VkStructureType sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    const void * pNext = nullptr;
    uint32_t setLayoutCount = 4;
    const VkDescriptorSetLayout * pSetLayouts = &vplciSetLayouts[0];
}
```

Prior to setting up the Push Constants, however, the pipeline layout needs to be told about the Push Constants:

```c
VkPipelineLayoutCreateInfo vplci;

vplci.pSetLayouts = DescriptorSetLayouts;
```

where:
- vpcr[0].offset = 0;
- vpcr[0].size = sizeof(glm::mat4);
- vpcr[0].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT | VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;

Prior to that, however, the pipeline layout needs to be told about the Push Constants:

```c
VkPipelineLayoutCreateInfo vplci;

vplci.pSetLayouts = DescriptorSetLayouts;
```

Where each arm is represented by:

```c
struct armArm1
{
    float     armScale;
    glm::vec3 armColor;
    glm::mat4 armMatrix;
};
```

An example of an arm structure:

```c
struct armArm1
{
    float     armScale;
    glm::vec3 armColor;
    glm::mat4 armMatrix;
};
```

An example of an arm structure:

```c
struct armArm1
{
    float     armScale;
    glm::vec3 armColor;
    glm::mat4 armMatrix;
};
```

An example of an arm structure:

```c
struct armArm1
{
    float     armScale;
    glm::vec3 armColor;
    glm::mat4 armMatrix;
};
```
Forward Kinematics:
You start with separate pieces, all defined in their own local coordinate system.

Hook the pieces together, change parameters, and things move.
(All young children understand this.)

Given the lengths and angles, where do the pieces move to?

Positioning part #1 with respect to ground:
1. Rotate by $\theta_1$
2. Translate by $T_{1/G}$

Positioning part #2 with respect to ground:
1. Rotate by $\theta_2$
2. Translate the length of part 1
3. Rotate by $\theta_1$
4. Translate by $T_{1/G}$

Why do we say it right-to-left?
We adopt the convention that the coordinates are multiplied on the right side of the matrix:

So the right-most transformation in the sequence multiplies the (x,y,z) first and the left-most transformation multiplies it last.
Positioning Part #3 With Respect to Ground

1. Rotate by $\Theta_3$
2. Translate the length of part 2
3. Rotate by $\Theta_2$
4. Translate the length of part 1
5. Rotate by $\Theta_1$
6. Translate by $T_{1/G}$

$$\left[ M_{3/G} \right] = \left[ T_{1/G} \right] \cdot \left[ R_{\Theta_3} \right] \cdot \left[ T_{2/G} \right] \cdot \left[ R_{\Theta_2} \right] \cdot \left[ T_{1/G} \right] \cdot \left[ R_{\Theta_1} \right]$$

Setup the Push Constant for the Pipeline Structure

VkPushConstantRange

- $voip[0].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT | VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT$
- $voip[0].offset = 0$
- $voip[0].size = \text{sizeof( struct arm )}$

In the Reset Function

struct arm

- $\text{Arm1.armMatrix} = \text{glm: mat4( )}$
- $\text{Arm1.armColor} = \text{glm: vec3(0.f, 1.f, 0.f)}$
- $\text{Arm1.armScale} = 6.f$

The constructor $\text{glm: mat4( )}$ produces an identity matrix. The actual transformation matrices will be set in $\text{UpdateScene( )}$.

In the UpdateScene Function

float rot1 = (float)Time;
float rot2 = 2.f * rot1;
float rot3 = 2.f * rot2;

$$\text{Arm1.armMatrix} = \text{glm: mat4( )}$$
$$\text{Arm1.armColor} = \text{glm: vec3(0.f, 1.f, 0.f)}$$
$$\text{Arm1.armScale} = 6.f$$

In the RenderScene Function Without Pipeline Barriers

VkBuffer

- $\text{buffers[1]} = \{ \text{MyVertexDataBuffer.buffer} \}$
- $\text{vkCmdBindVertexBuffers(CommandBuffers[nextImageIndex], 0, 1, buffers, offsets)}$
- $\text{vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm1)}$
- $\text{vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance)}$
- $\text{vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm2)}$
- $\text{vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance)}$
- $\text{vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm3)}$
- $\text{vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance)}$

But, the problem is that

1. The $\text{vkCmdDraws}$ must not start until the $\text{vkCmdPushConstants}$ are done, and
2. The $\text{vkCmdPushConstants}$ must not start until the $\text{vkCmdDraws}$ are done

This is the type of problem that Pipeline Barriers were meant to solve.

Setting Up Global Memory Pipeline Barriers

VkMemoryBarrier

- $\text{vkCmdPipelineBarrier(commandBuffer, VK_COMMAND_BARRIER_BY_REGION_BIT, 0, 0, 0, 0, 0, 0, 0)}$

$\text{GraphicsPipelineLayout}$
In the **RenderScene** Function

```cpp
VkBuffer buffers[1] = { MyVertexDataBuffer.buffer };    vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, buffers, offsets );

vkCmdPushConstants( CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm1 );
vkCmdPipelineBarrier(CommandBuffers[nextImageIndex], srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 1, IN vmb, 0, nullptr, 0, nullptr );

vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );

vkCmdPushConstants( CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm2 );
vkCmdPipelineBarrier(CommandBuffers[nextImageIndex], srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 1, IN vmb, 0, nullptr, 0, nullptr );

vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );

vkCmdPushConstants( CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm3 );
vkCmdPipelineBarrier(CommandBuffers[nextImageIndex], srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 1, IN vmb, 0, nullptr, 0, nullptr );

vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
```

In the **Vertex Shader**

```cpp
layout( push constant ) uniform arm {
  mat4 armMatrix;
  vec3 armColor;
  float armScale; // scale factor in x
} RobotArm;

layout( location = 0 ) in vec3 aVertex;

. . .

vec3 bVertex = aVertex; // arm coordinate system is [-1., 1.] in X
bVertex.x += 1.; // now is [0., 2.]

bVertex.x /= 2.; // now is [0., 1.]

bVertex.x *= (RobotArm.armScale); // now is [0., RobotArm.armScale]

bVertex = vec3( RobotArm.armMatrix * vec4( bVertex, 1. ) );

. . .

gl_Position = PVM * vec4( bVertex, 1. ); // Projection * Viewing * Modeling matrices
```

---

**Setting Up Buffer Memory Pipeline Barriers**

```cpp
vkBufferMemoryBarrier vbmb;
vbmb.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
vbmb.pNext = nullptr;
vbmb.srcAccessMask = dstAccessMask = vbmb.srcQueueFamilyIndex = dstQueueFamilyIndex = vbmb.buffer = vbmb.offset = vbmb.size =

vkCmdPipelineBarrier( commandBuffer, srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 0, NULL, 1, IN &vbmb, 0, nullptr );
```

**Setting Up Image Memory Pipeline Barriers**

```cpp
VkImageMemoryBarrier vimb;
vimb.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
vimb.pNext = nullptr;
vimb.srcAccessMask = dstAccessMask = vimb.oldLayout = newLayout = vimb.srcQueueFamilyIndex = dstQueueFamilyIndex = vimb.image = vimb.subResourceRange =

vkCmdPipelineBarrier( commandBuffer, srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 0, NULL, 0, NULL, 1, IN &vimb );
```