Push Constants

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.
Push Constants

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

```cpp
layout( push_constant ) uniform matrix
{
  mat4 modelMatrix;
} Matrix;
```

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

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

where:

- `stageFlags` are or'ed 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`.

Setting up the Push Constants for the Pipeline Structure

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

```cpp
VkPushConstantRange vpcr[1];

vpcr[0].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT |
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;

vpcr[0].offset = 0;

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

VkPipelineLayoutCreateInfo vplci;

vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;

vplci.pNext = nullptr;

vplci.flags = 0;

vplci.setLayoutCount = 4;

vplci.pSetLayouts = DescriptorSetLayouts;

result = vkCreatePipelineLayout( LogicalDevice, IN &vplci, PALLOCATOR, OUT &GraphicsPipelineLayout );
```
Creating a Pipeline

- Shaders
- Vertex Input State
- Input Assembly State
- Rasterization State
- Depth Stencil State
- Dynamic State
- Pipeline layout
- Render Pass
- basePipelineHandle
- basePipelineIndex

An Robotic Example using Push Constants

A robotic animation (i.e., a hierarchical transformation system)

Where each arm is represented by:

```c
struct arm {
    glm::mat4 armMatrix;
    glm::vec3 armColor;
    float armScale; // scale factor in x
};
```

`struct armArm1;`  
`struct armArm2;`  
`struct armArm3;`  

```c
struct armArm1;  
struct armArm2;  
struct armArm3;
```
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)
Forward Kinematics: Given the Lengths and Angles, Where do the Pieces Move To?

Locations?

1. Rotate by $\Theta_1$
2. Translate by $T_{1/G}$

Write it

$$[M_{1/G}] = [T_{1/G}] * [R_{\Theta_1}]$$

Say it
Why Do We Say it Right-to-Left?

We adopt the convention that the coordinates are multiplied on the right side of the matrix:

\[
\begin{bmatrix}
    x' \\
    y' \\
    z'
\end{bmatrix} =
\begin{bmatrix}
    A & B & C & D \\
    E & F & G & H \\
    I & J & K & L
\end{bmatrix}
\begin{bmatrix}
    x \\
    y \\
    z \\
    1
\end{bmatrix}
\]

So the right-most transformation in the sequence multiplies the \((x, y, z, 1)\) \textit{first} and the left-most transformation multiples it \textit{last}.

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 \(T_{1/G}\)

\[
\begin{bmatrix}
    M_{2/G} \\
\end{bmatrix} =
\begin{bmatrix}
    T_{1/G} \\
\end{bmatrix}
\begin{bmatrix}
    R_{\Theta_1} \end{bmatrix}
\begin{bmatrix}
    T_{2/1} \end{bmatrix}
\begin{bmatrix}
    R_{\Theta_2} \end{bmatrix}
\]

\[
\begin{bmatrix}
    M_{2/G} \\
\end{bmatrix} =
\begin{bmatrix}
    M_{1/G} \\
\end{bmatrix}
\begin{bmatrix}
    M_{2/1} \\
\end{bmatrix}
\]
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 $T_{1/G}$

\[
\begin{bmatrix}
M_{3/G}
\end{bmatrix} = \begin{bmatrix}
T_{1/G}
\end{bmatrix} \ast \begin{bmatrix}
R_{\Theta_1}
\end{bmatrix} \ast \begin{bmatrix}
T_{2/1}
\end{bmatrix} \ast \begin{bmatrix}
R_{\Theta_2}
\end{bmatrix} \ast \begin{bmatrix}
T_{3/2}
\end{bmatrix} \ast \begin{bmatrix}
R_{\Theta_3}
\end{bmatrix}
\]

Say it

In the Reset Function

```cpp
struct arm
struct arm
struct arm

Arm1.armMatrix = glm::mat4();
Arm1.armColor  = glm::vec3(0.f, 1.f, 0.f);
Arm1.armScale  = 6.f;

Arm2.armMatrix = glm::mat4();
Arm2.armColor  = glm::vec3(1.f, 0.f, 0.f);
Arm2.armScale  = 4.f;

Arm3.armMatrix = glm::mat4();
Arm3.armColor  = glm::vec3(0.f, 0.f, 1.f);
Arm3.armScale  = 2.f;
```

The constructor `glm::mat4()` produces an identity matrix. The actual transformation matrices will be set in `UpdateScene()`. 
Setup the Push Constant for the Pipeline Structure

```c
VkPushConstantRange vpcr[1];
    vpcr[0].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT |
                        VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
    vpcr[0].offset = 0;
    vpcr[0].size = sizeof(struct arm);

VkPipelineLayoutCreateInfo vplci;
    vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    vplci.pNext = nullptr;
    vplci.flags = 0;
    vplci.setLayoutCount = 4;
    vplci.pSetLayouts = DescriptorSetLayouts;
    vplci.pushConstantRangeCount = 1;
    vplci.pPushConstantRanges = vpcr;

result = vkCreatePipelineLayout(LogicalDevice, IN &vplci, PALLOCATOR, 
OUT &GraphicsPipelineLayout);
```

In the UpdateScene Function

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

glm::vec3 zaxis = glm::vec3(0., 0., 1.);

glm::mat4 m1g = glm::mat4();
    m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));
    m1g = glm::rotate(m1g, rot1, zaxis);

glm::mat4 m21 = glm::mat4();
    m21 = glm::translate(m21, glm::vec3(2.*Arm1.armScale, 0., 0.));
    m21 = glm::rotate(m21, rot2, zaxis);

glm::mat4 m32 = glm::mat4();
    m32 = glm::translate(m32, glm::vec3(2.*Arm2.armScale, 0., 0.));
    m32 = glm::rotate(m32, rot3, zaxis);
    Arm1.armMatrix = m1g; // m1g
    Arm2.armMatrix = m1g * m21; // m2g
    Arm3.armMatrix = m1g * m21 * m32; // m3g
```
In the RenderScene Function Without Pipeline Barriers

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

vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm2);
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);

vkCmdPushConstants(CommandBuffers[nextImageIndex], GraphicsPipelineLayout, VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm3);
vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);
```

But, the problem is that
1. The vkCmdDraws must not start until the vkCmdPushConstants are done, and
2. The vkCmdPushConstants must not start until the vkCmdDraws are done

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

Setting Up Global Memory Pipeline Barriers

```c
VkMemoryBarrier vmb;
    vmb.sType = VK_STRUCTURE_TYPE_MEMORY_BARRIER;
    vmb.pNext = nullptr;
    vmb.srcAccessMask =
    vmb.dstAccessMask =

vkCmdPipelineBarrier(commandBuffer,
    srcStageMask,  
    dstStageMask,  
    VK_DEPENDENCY_BY_REGION_BIT,  
    1,  
    &vmb,
    0,  
    nullptr,
    0,  
    nullptr);
```
### Setting Up Buffer Memory Pipeline Barriers

```cpp
VkBufferMemoryBarrier vbmb;
    vbmb.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
    vbmb.pNext = nullptr;
    vbmb.srcAccessMask =
    vbmb.dstAccessMask =
    vbmb.srcQueueFamilyIndex =
    vbmb.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 =
    vimb.dstAccessMask =
    vimb.oldLayout =
    vimb.newLayout =
    vimb.srcQueueFamilyIndex =
    vimb.dstQueueFamilyIndex =
    vimb.image =
    vimb.subResourceRange =
    vkCmdPipelineBarrier( commandBuffer,
                       srcStageMask,
                       dstStageMask,
                       VK_DEPENDENCY_BY_REGION_BIT,
                       0, NULL,
                       1, IN &vimb );
```
In the `RenderScene` Function

```c
VkBuffer buffers[1] = { MyVertexDataBuffer.buffer; }

vkCmdBindVertexBuffer(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 vm, 0, nullptr, 0, nullptr );

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

```c
vkCmdPipelineBarrier(CommandBuffers[nextImageIndex], srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 1, IN vm, 0, nullptr, 0, nullptr );

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 vm, 0, nullptr, 0, nullptr );

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

```c
vkCmdPipelineBarrier(CommandBuffers[nextImageIndex], srcStageMask, dstStageMask, VK_DEPENDENCY_BY_REGION_BIT, 1, IN vm, 0, nullptr, 0, nullptr );

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 vm, 0, nullptr, 0, nullptr );

vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
```
In the Vertex Shader

```glsl
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
```

Oregon State University
Computer Graphics

mjb – October 5, 2018