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.

Creating a Pipeline

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

```glsl
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:

```c
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:

```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(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;

vplci.pushConstantRangeCount = 1;

vplci.pPushConstantRanges = vpcr;

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

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
};
```

1
2
3

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?

Positions?

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

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

Why Do We Say it Right-to-Left?

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

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = [M_{1/G}] \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = [T_{1/G}] * [R_{\Theta_1}] * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

So the right-most transformation in the sequence multiplies the $(x,y,z,1)$ first and the left-most transformation multiplies it last.

Positioning Part #1 With Respect to Ground

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

$$[M_{2/G}] = [T_{1/G}] * [R_{\Theta_1}] * [T_{2/1}] * [R_{\Theta_2}]$$

$$[M_{2/G}] = [M_{1/G}] * [M_{2/1}]$$

Positioning Part #2 With Respect to Ground
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 $T_{1/G}$

\[
\begin{align*}
[M_{3/G}] &= [T_{1/G}] \cdot [R_{\Theta_1}] \cdot [T_{2/1}] \cdot [R_{\Theta_2}] \cdot [T_{3/2}] \cdot [R_{\Theta_3}] \\
[M_{3/G}] &= [M_{1/G}] \cdot [M_{2/1}] \cdot [M_{3/2}]
\end{align*}
\]

Write it
Say it

In the Reset Function

struct arm Arm1;
struct arm Arm2;
struct arm Arm3;

... 

\[
\begin{align*}
\text{Arm1.armMatrix} &= \text{glm::mat4(1.1);  // identity} \\
\text{Arm1.armColor} &= \text{glm::vec3(0.f, 1.f, 0.f);} \\
\text{Arm1.armScale} &= 6.f;
\end{align*}
\]

\[
\begin{align*}
\text{Arm2.armMatrix} &= \text{glm::mat4(1.1);} \\
\text{Arm2.armColor} &= \text{glm::vec3(1.f, 0.f, 0.f);} \\
\text{Arm2.armScale} &= 4.f;
\end{align*}
\]

\[
\begin{align*}
\text{Arm3.armMatrix} &= \text{glm::mat4(1.1);} \\
\text{Arm3.armColor} &= \text{glm::vec3(0.f, 0.f, 1.f);} \\
\text{Arm3.armScale} &= 2.f;
\end{align*}
\]

The constructor \texttt{glm::mat4(1.1)} produces an identity matrix. The actual transformation matrices will be set in UpdateScene().

In the UpdateScene Function

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

\[
\begin{align*}
\text{glm::vec3 zaxis} &= \text{glm::vec3(0., 0., 1.);} \\
\text{glm::mat4 m1g} &= \text{glm::mat4(1.1);} & \text{identity} \\
\text{m1g} &= \text{glm::translate(m1g, glm::vec3(0., 0., 0.));} \\
\text{m1g} &= \text{glm::rotate(m1g, rot1, zaxis);} & \text{[T][R]} \\
\text{glm::mat4 m21} &= \text{glm::mat4(1.1);} & \text{identity} \\
\text{m21} &= \text{glm::translate(m21, glm::vec3(2.*Arm1.armScale, 0., 0.));} \\
\text{m21} &= \text{glm::rotate(m21, rot2, zaxis);} & \text{[T][R]} \\
\text{m21} &= \text{glm::translate(m21, glm::vec3(0., 0., 2.));} & \text{z-offset from previous arm}
\end{align*}
\]

\[
\begin{align*}
\text{glm::mat4 m32} &= \text{glm::mat4(1.1);} & \text{identity} \\
\text{m32} &= \text{glm::translate(m32, glm::vec3(2.*Arm2.armScale, 0., 0.));} \\
\text{m32} &= \text{glm::rotate(m32, rot3, zaxis);} & \text{[T][R]} \\
\text{m32} &= \text{glm::translate(m32, glm::vec3(0., 0., 2.));} & \text{z-offset from previous arm}
\end{align*}
\]

\[
\begin{align*}
\text{Arm1.armMatrix} &= \text{m1g}; & \text{m1g} \\
\text{Arm2.armMatrix} &= \text{m1g} \cdot \text{m21}; & \text{m2g} \\
\text{Arm3.armMatrix} &= \text{m1g} \cdot \text{m21} \cdot \text{m32}; & \text{m3g}
\end{align*}
\]
In the RenderScene Function

```cpp
def RenderScene():
    VkBuffer buffers[1] = { MyVertexBuffer.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)
```

The strategy is to draw each link using the same vertex buffer, but modified with a unique color, length, and matrix transformation.

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
```