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.

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`.

An Robotic Example using Push Constants

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

Where each arm is represented by:

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

struct armArm1;
struct armArm2;
struct armArm3;
Forward Kinematics:
You Start with Separate Pieces, all Defined in their Own Local Coordinate System

![Diagram of separate pieces in local coordinate systems]

Hook the Pieces Together, Change Parameters, and Things Move
(All Young Children Understand This)

![Diagram showing how pieces are hooked together]

Given the Lengths and Angles, Where do the Pieces Move To?

![Diagram showing movement of pieces]

Why Do We Say it Right-to-Left?

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

\[
\begin{pmatrix}
A \\
B \\
C \\
D \\
y' \\
y \\
z' \\
z
\end{pmatrix} = \begin{pmatrix}
\theta_1 \\
\theta_2
\end{pmatrix} * \begin{pmatrix}
R_1 \\
R_2
\end{pmatrix}
\]

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

Positioning Part #1 With Respect to Ground

1. Rotate by \(\theta_1\)
2. Translate by \(T_{1/G}\)

\[
M_{1/G} = T_{1/G} * R_{\theta_1}
\]

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 \(T_{1/G}\)

\[
M_{2/G} = T_{1/G} * R_{\theta_1} * T_{2/1} * R_{\theta_2}
\]

\[
M_{2/G} = M_{1/G} * M_{2/1}
\]
Positioning Part #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 \( T_{\frac{1}{G}} \)

\[
[M_{G/3}] = [T_{1/G}] \cdot [R_{31}] \cdot [T_{21}] \cdot [R_{21}] \cdot [T_{12}] \cdot [R_{12}]
\]

Write it

Say it

In the Reset Function

\[
\begin{align*}
\text{Arm1.armMatrix} &= \text{glm::mat4()} \\
\text{Arm1.armColor} &= \text{glm::vec3(0.1, 0.3, 0.1)} \\
\text{Arm1.armScale} &= 6.0f \\
\text{Arm2.armMatrix} &= \text{glm::mat4()} \\
\text{Arm2.armColor} &= \text{glm::vec3(1.0, 0.5, 0.1)} \\
\text{Arm2.armScale} &= 4.0f \\
\text{Arm3.armMatrix} &= \text{glm::mat4()} \\
\text{Arm3.armColor} &= \text{glm::vec3(0.1, 0.5, 1.0)} \\
\text{Arm3.armScale} &= 2.1f
\end{align*}
\]

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

In the UpdateScene Function

\[
\begin{align*}
\text{float rot1} &= (\text{float})\text{Time} \\
\text{float rot2} &= 2.0f \times \text{rot1} \\
\text{float rot3} &= 2.0f \times \text{rot2} \\
\text{glm::vec3 zaxis} &= \text{glm::vec3}(0., 0., 1.) \\
\text{glm::mat4 m1g} &= \text{glm::mat4()} \\
\text{m1g} &= \text{glm::translate(m1g, glm::vec3(0., 0., 0.))} \\
\text{m1g} &= \text{glm::rotate(m1g, rot1, zaxis)} \\
\text{glm::mat4 m21} &= \text{glm::mat4()} \\
\text{m21} &= \text{glm::translate(m21, glm::vec3(2.0f*Arm1.armScale, 0., 0.))} \\
\text{m21} &= \text{glm::rotate(m21, rot2, zaxis)} \\
\text{m21} &= \text{glm::translate(m21, glm::vec3(0., 0., 2.))} \\
\text{glm::mat4 m32} &= \text{glm::mat4()} \\
\text{m32} &= \text{glm::translate(m32,glm::vec3(2.0f*Arm2.armScale, 0., 0.))} \\
\text{m32} &= \text{glm::rotate(m32, Arm3.armScale, glm::vec3(0., 0., 2.))} \\
\text{Arm1.armMatrix} &= \text{m1g} \\
\text{Arm2.armMatrix} &= \text{m1g} \times \text{m21} \\
\text{Arm3.armMatrix} &= \text{m1g} \times \text{m21} \times \text{m32}
\end{align*}
\]

In the RenderScene Function

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

In the Vertex Shader

\[
\begin{align*}
\text{layout} \text{push constant | uniform arm } \{ \\
\text{mat4 armMatrix; } \\
\text{vec3 armColor; } & \text{ scale factor in x} \\
\text{float armScale}; \} \text{ RobotArm; } \\
\text{layout(location = 0) in vec3 aVertex; } \text{ RobotArm; }
\end{align*}
\]

\[
\begin{align*}
\text{vec3 bVertex} &= \text{aVertex}; \text{ RobotArm; }
\text{bVertex.x} &= \text{bVertex.x} + 1.; \text{ RobotArm; }
\text{bVertex.x} &= \text{bVertex.x} / 2.; \text{ RobotArm; }
\text{bVertex.x} &= \text{bVertex.x} \times (\text{RobotArm.armScale}); \text{ RobotArm; }
\text{bVertex} &= \text{vec3(\text{RobotArm.armMatrix} \times \text{vec4(bVertex, 1.)})}; \text{ RobotArm; }
\text{gl_Position} &= \text{PVM} \times \text{vec4(bVertex, 1.)}; \text{ RobotArm; }
\end{align*}
\]