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

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

```cpp
struct armArm1 { float armScale; // scale factor in x glm::vec3 armColor; glm::mat4 armMatrix; }

struct armArm2 { float armScale; // scale factor in y glm::vec3 armColor; glm::mat4 armMatrix; }

struct armArm3 { float armScale; // scale factor in z glm::vec3 armColor; glm::mat4 armMatrix; }

// where:
// armScale is a float // armColor is a glm::vec3 // armMatrix is a glm::mat4
// armArm1, armArm2, armArm3 are the arms

struct arm { armArm1 arm1; armArm2 arm2; armArm3 arm3; }
```

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:

In an effect 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:
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?

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 #1 With Respect to Ground

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

Write it

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

Say it

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 $T_{G1}$

Write it

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

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

Say it
3. Rotate by $\Theta$
4. Translate by $T$
5. Rotate by $\Theta$
6. Translate by $T$

\[
M_{S,G} = \left[ T_{1,G} \right] \left[ R_{G,1} \right] \left[ T_{2,1} \right] \left[ R_{G,2} \right] \left[ T_{3,2} \right] \left[ R_{G,3} \right]
\]

\[
M_{1,G} = \left[ M_{1,G} \right] \left[ M_{2,1} \right] \left[ M_{3,2} \right]
\]

---

In the Reset Function

\[
\begin{align*}
\text{Arm1.armMatrix} &= \text{glm::mat4}(1, 1), \\
\text{Arm1.armColor} &= \text{glm::vec3}(0, 1, 0, 1), \\
\text{Arm1.armScale} &= 6.0f;
\end{align*}
\]

\[
\begin{align*}
\text{Arm2.armMatrix} &= \text{glm::mat4}(1, 1), \\
\text{Arm2.armColor} &= \text{glm::vec3}(1, 0, 0, 0), \\
\text{Arm2.armScale} &= 4.0f;
\end{align*}
\]

\[
\begin{align*}
\text{Arm3.armMatrix} &= \text{glm::mat4}(1, 1), \\
\text{Arm3.armColor} &= \text{glm::vec3}(0, 0, 0, 1), \\
\text{Arm3.armScale} &= 2.0f;
\end{align*}
\]

---

The constructor $\text{glm::mat4}(1, 1)$ produces an identity matrix. The actual transformation matrices will be set in $\text{UpdateScene}()$.

---

In the UpdateScene Function

\[
\begin{align*}
\text{float rot1} &= (\text{float}) \text{Time}; \\
\text{float rot2} &= 2.0f \times \text{rot1}; \\
\text{float rot3} &= 2.0f \times \text{rot2};
\end{align*}
\]

\[
\begin{align*}
\text{glm::vec3} \text{zaxis} &= \text{glm::vec3}(0, 0, 1); \\
\text{glm::mat4} \text{m1g} &= \text{glm::mat4}(1, 1), 0 \text{id}; \\
\text{glm::mat4} \text{m2g} &= \text{glm::mat4}(1, 1), 0 \text{id}; \\
\text{glm::mat4} \text{m3g} &= \text{glm::mat4}(1, 1), 0 \text{id};
\end{align*}
\]

---

In the Vertex Shader

```
layout( push_constant ) uniform arm

vec3  armColor; 
vec4  armMatrix;
```

---

In the RenderScene Function

```
struct arm

struct Arm1
struct Arm2
struct Arm3

Arm1.armMatrix = glm::mat4(1, 1); 
Arm1.armColor = glm::vec3(0, 1, 0, 1); 
Arm1.armScale = 6.0f;

Arm2.armMatrix = glm::mat4(1, 1); 
Arm2.armColor = glm::vec3(1, 0, 0, 0); 
Arm2.armScale = 4.0f;

Arm3.armMatrix = glm::mat4(1, 1); 
Arm3.armColor = glm::vec3(0, 0, 0, 1); 
Arm3.armScale = 2.0f;
```