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, such as matrix transformation matrices. This is a good feature, since Vulkan, at times, makes it cumbersome to send changes to the graphics.

By “small”, Vulkan specifies that there will be at least 128 bytes that can be used, 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 do not live in their own GPU memory. They are actually included inside the Vulkan graphics pipeline data structure.

Push Constants

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

Where each arm is represented by:

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

struct Arm1, Arm2, Arm3;
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 $T_{1/G}$

Code it

Say it

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

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 $T_{1/G}$

Code it

Say it

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

$M_{2/G} = [M_{1/G}] \cdot [M_{2/G}]$

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 \( T_{WG} \)

\[
\begin{align*}
[M_{3G}] &= [T_{1G}] \cdots [R_{g2}] \cdots [T_{2G}]) \cdots [R_{g3}] \\
[M_{BG}] &= [M_{1G}] \cdots [M_{2G}] \cdots [M_{3G}]
\end{align*}
\]

In the Reset Function

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

- Arm1.armMatrix = glm::mat4(1.); // rotation for arm1, in radians
- Arm1.armColor = glm::vec3(0.f, 1.f, 0.f); // green
- Arm1.armScale = 6.f;

- Arm2.armMatrix = glm::mat4(1.); // rotation for arm2, in radians
- Arm2.armColor = glm::vec3(1.f, 0.f, 1.f); // red
- Arm2.armScale = 4.f;

- Arm3.armMatrix = glm::mat4(1.); // rotation for arm3, in radians
- Arm3.armColor = glm::vec3(0.f, 0.f, 1.f); // blue
- Arm3.armScale = 2.f;

The constructor `glm::mat4(1.)` produces an identity matrix. The actual transformation matrices will be set in `UpdateScene()`. 

Set the Push Constant for the Graphics Pipeline Data Structure

```cpp
VkPushConstantRange

vpoclstageFlags = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
vpocloffset = 0;
vpocl.size = sizeof(struct arm);

VkPipelineLayoutCreateInfo

vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
vplci.pNext = nullptr;
vplci.pPipelineLayouts = DescriptorSetLayouts;
vplci.setLayoutCount = 5;
vplci.pSetLayouts = DescriptorSetLayouts;
vplci.flags = 0;

std::vector<VkPushConstantRange> vpcr;
vpcr[0].size = sizeof(struct arm);
vpcr[0].offset = 0;
vpcr[0].stageFlags = VK_SHADER_STAGE_ALL;

vplci.pPushConstantRanges = vpcr;
```

In the UpdateSceneFunction

```cpp
float rot1 = (float)(2.*M_PI*Time); // rotation for arm1, in radians
float rot2 = 2.f * rot1; // rotation for arm2, in radians
float rot3 = 2.f * rot2; // rotation for arm3, in radians

glm::vec3 zaxis = glm::vec3(0., 0., 1.);
float rot3 = 2.f * rot2; // rotation for arm3, in radians
float rot2 = 2.f * rot1; // rotation for arm2, in radians
float rot1 = (float)(2.*M_PI*Time); // rotation for arm1, in radians

Arm1.armMatrix = glm::rotate(m1g, rot1, zaxis); // rotation for arm1, in radians
Arm1.armMatrix = glm::translate(m1g, glm::vec3(0., 0., 0.)); // offset from previous arm

Arm2.armMatrix = glm::rotate(m2g, rot2, zaxis); // rotation for arm2, in radians
Arm2.armMatrix = glm::translate(m2g, glm::vec3(0., 0., 2.)); // offset from previous arm

Arm3.armMatrix = glm::rotate(m3g, rot3, zaxis); // rotation for arm3, in radians
Arm3.armMatrix = glm::translate(m3g, glm::vec3(0., 0., 2.)); // offset from previous arm
```

In the RenderScene Function

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

In the Vertex Shader

```cpp
layout (location = 0 ) in vec3 aVertex;
layout (push_constant) uniform arm Arm1;
layout (push_constant) uniform arm Arm2;
layout (push_constant) uniform arm Arm3;

Arm1.armMatrix = m1g; // m1g
Arm2.armMatrix = m2g; // m2g
Arm3.armMatrix = m3g; // m3g

Arm1.armScale  = 2.f;
Arm1.armColor  = glm::vec3( 0.f, 1.f, 0.f ); // green
Arm1.armMatrix = glm::mat4( 1. );

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

Arm3.armScale  = 6.f;
Arm3.armColor  = glm::vec3( 0.f, 0.f, 1.f ); // blue
Arm3.armMatrix = glm::mat4( 1. );

float   armScale;         // scale factor in x
vec3  armColor;
mat4  armMatrix;
```

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