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 application side, push constants are pushed at the shaders by binding them 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.

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.

**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:

```c
layout( push_constant ) uniform matrix

Matrices;
```

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

```c
dgCmdPushConstants( 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:

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

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

Forward Kinematics:
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?

Why Do We Say it Right-to-Left?
We adopt the convention that the coordinates are multiplied on the right side of the matrix:

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

So the right-most transformation in the sequence multiplies the (x,y,z,1) first and the left-most transformation multiples 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 \( T_{1/G} \)

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

Say it
1. Rotate by \( \Theta \)
2. Translate the length of part 2
3. Rotate by \( \Theta \)
4. Translate the length of part 1
5. Rotate by \( \Theta \)
6. Translate by \( \overline{1}_{1/2} \)

\[
[M_{3/G}] = [T_{3/G}] * [R_{3}] * [T_{2/G}] * [R_{2}] * [T_{1/G}] * [R_{1}]
\]

\[
[M_{3/G}] = [M_{1/G}] * [M_{2/G}] * [M_{3/G}]
\]

**Positioning Part #3 With Respect to Ground**

**In the Reset Function**

- struct arm
  - Arm1:
    - struct arm
      - Arm1:
        - struct arm
          - Arm1:

  - Arm2:
    - struct arm
      - Arm2:
        - struct arm
          - Arm2:

  - Arm3:
    - struct arm
      - Arm3:
        - struct arm
          - Arm3:

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

**Setup the Push Constant for the Pipeline Structure**

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

VkPipelineLayoutCreateInfo
vplci.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
vplci.pSetLayouts = DescriptorSetLayouts;
vplci.setLayoutCount = 4;
vplci.pNext = nullptr;
vplci.pPushConstantRanges = &vpcr[0];

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

**In the Reset Function**

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

```c
glm::mat4 m1g = glm::rotate(m1g, rot1, zaxis); // [T][R]
```

**In the UpdateScene Function**

```c
glm::mat4 m21 = glm::translate(m21, glm::vec3(0., 0., 2.)); // z-offset from previous arm
```

**In the RenderScene Function**

```c
layout(push_constant) uniform arm

struct arm;
```

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**

```c
layout(location = 0) in vec3 aVertex;
```

In the Vertex Shader

```c
layout(push_constant) uniform arm

struct arm:

vec3 armColor;
float armScale;
mat4 armMatrix;
```

```c
vec3 bVertex = aVertex; // arm coordinate system is [-1., 1.] in X
```

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

```c
float rot1 = (float)Time;
```

```c
float rot2 = 2.f * rot1;
```

```c
float rot3 = 2.f * rot2;
```

```c
vec3 bVertex = aVertex; // arm coordinate system is [-1., 1.] in X
```

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

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

```c
bVertex.x *= 2.; // now is [0., 1.]
```

```c
bVertex.x += 1.; // now is [0., 2.]
```

```c
vec3 bVertex = aVertex; // arm coordinate system is [-1., 1.] in X
```

```c
m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));
```

```c
m1g = glm::rotate(m1g, zaxis); // [T][R]
```

```c
m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));
```

```c
glm::mat4 m1g = glm::mat4();
```

In the Vertex Shader

```c
vec3 bVertex = aVertex; // arm coordinate system is [-1., 1.] in X
```

```c
void buffer( ) = (MyVertexDataBuffer.buffer);
```

```c
void Buffer( ) = (MyVertexDataBuffer.buffer);
```

```c
void Buffer( ) = (MyVertexDataBuffer.buffer);
```

```c
void Buffer( ) = (MyVertexDataBuffer.buffer);
```

```c
void Buffer( ) = (MyVertexDataBuffer.buffer);
```

```c
void Buffer( ) = (MyVertexDataBuffer.buffer);
```

```c
void Buffer( ) = (MyVertexDataBuffer.buffer);
```

```c
void Buffer( ) = (MyVertexDataBuffer.buffer);
```

```c
void Buffer( ) = (MyVertexDataBuffer.buffer);
```

```c
void Buffer( ) = (MyVertexDataBuffer.buffer);
```

```c
void Buffer( ) = (MyVertexDataBuffer.buffer);
```

```c
void Buffer( ) = (MyVertexDataBuffer.buffer);
```