Push Constants

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:

```
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:

```
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`.

Prior to that, however, the pipeline layout needs to be told about the Push Constants:

```
VkPushConstantRange
vpcr[0].stageFlags = VK_PIPELINE_STAGE_VERTEX_SHADER_BIT |
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
```

```
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, PALLOCATOR, OUT &GraphicsPipelineLayout );
```

Setting up the Push Constants for the Pipeline Structure

Prior to that, however, the pipeline layout needs to be told about the Push Constants:
Creating a Pipeline

- VkGraphicsPipelineCreateInfo
  - Shaders
  - VertexInput State
  - InputAssembly State
  - Tesselation State
  - Viewport State
  - Rasterization State
  - MultiSample State
  - DepthStencil State
  - ColorBlend State
  - Dynamic State
  - Pipeline layout
  - RenderPass

- VkPipelineShaderStageCreateInfo
- VkPipelineVertexInputStateCreateInfo
- VkVertexInputBindingDescription
- VkViewportStateCreateInfo
- Viewport: x, y, w, h, minDepth, maxDepth
- Scissor: offset, extent
- VkPipelineRasterizationStateCreateInfo
- CullMode
- PolygonMode
- FrontFace
- LineWidth
- VkPipelineInputAssemblyStateCreateInfo
- Topology
- VkVertexInputAttributeDescription
- Binding, Stride, InputRate, Location
- VkPipelineDepthStencilStateCreateInfo
- VkPipelineColorBlendStateCreateInfo
- DepthTestEnable
- DepthWriteEnable
- DepthCompareOp
- StencilTestEnable
- StencilOpStateFront
- StencilOpStateBack
- BlendEnable
- SrcColorBlendFactor, DstColorBlendFactor
- ColorBlendOp
- SrcAlphaBlendFactor, DstAlphaBlendFactor
- AlphaBlendOp
- ColorWriteMask
- VkPipelineDynamicStateCreateInfo
- VkCreateGraphicsPipeline

An Robotic Example using Push Constants

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

Where each arm is represented by:

```
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

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?

Location?

1

2

3

θ1

θ2

θ3

Ground

Why Do We Say it Right-to-Left?

Write it

\[
[M_{1/G}] = [T_{1/G}] \ast [R_{θ1}]
\]

Say it

Positioning Part #1 With Respect to Ground

1. Rotate by θ1
2. Translate by \( T_{1/G} \)

Positioning Part #2 With Respect to Ground

1. Rotate by θ2
2. Translate the length of part 1
3. Rotate by θ1
4. Translate by \( T_{1/G} \)

Write it

\[
[M_{2/G}] = [T_{1/G}] \ast [R_{θ1}] \ast [T_{2/G}] \ast [R_{θ2}]
\]

Say it

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

\[
[M_{3/G}] = [T_{1/G}] * [R_{\Theta_1}] * [T_{2/1}] * [R_{\Theta_2}] * [T_{3/2}] * [R_{\Theta_3}]
\]

In the Reset Function

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

... 
Arm1.armMatrix = glm::mat4(1);
Arm1.armColor  = glm::vec3(0.f, 1.f, 0.f);
Arm1.armScale  = 6.f;
Arm2.armMatrix = glm::mat4(1);
Arm2.armColor  = glm::vec3(1.f, 0.f, 0.f);
Arm2.armScale  = 4.f;
Arm3.armMatrix = glm::mat4(1);
Arm3.armColor  = glm::vec3(0.f, 0.f, 1.f);
Arm3.armScale  = 2.f;

The constructor glm::mat4(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;

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

m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));
m1g = glm::rotate(m1g, rot1, zaxis);

m21 = glm::translate(m21, glm::vec3(2.*Arm1.armScale, 0., 0.));
m21 = glm::rotate(m21, rot2, zaxis);
m21 = glm::translate(m21, glm::vec3(0., 0., 2.));
m32 = glm::translate(m32, glm::vec3(2.*Arm2.armScale, 0., 0.));
m32 = glm::rotate(m32, rot3, zaxis);
m32 = glm::translate(m32, glm::vec3(0., 0., 2.));

Arm1.armMatrix = m1g;  // m1g
Arm2.armMatrix = m1g * m21;  // m2g
Arm3.armMatrix = m1g * m21 * m32;  // m3g

Setup the Push Constant for the Pipeline Structure

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(struct arm);

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, PALLOCATOR, OUT &GraphicsPipelineLayout);
In the RenderScene Function

```c
VkBuffer buffers[1] = { MyVertexDataBuffer.buffer };  
vCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, buffers, offsets );  
vCmdPushConstants( CommandBuffers[nextImageIndex], GraphicsPipelineLayout,  
    VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm1 );  
vCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );  
vCmdPushConstants( CommandBuffers[nextImageIndex], GraphicsPipelineLayout,  
    VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm2 );  
vCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );  
vCmdPushConstants( CommandBuffers[nextImageIndex], GraphicsPipelineLayout,  
    VK_SHADER_STAGE_ALL, 0, sizeof(struct arm), (void *)&Arm3 );  
vCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
```

In the Vertex Shader

```c
layout( push_constant ) uniform arm  
{
    mat4 armMatrix;
    vec3 armColor;  
    float armScale;  
} 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
```