Push Constants

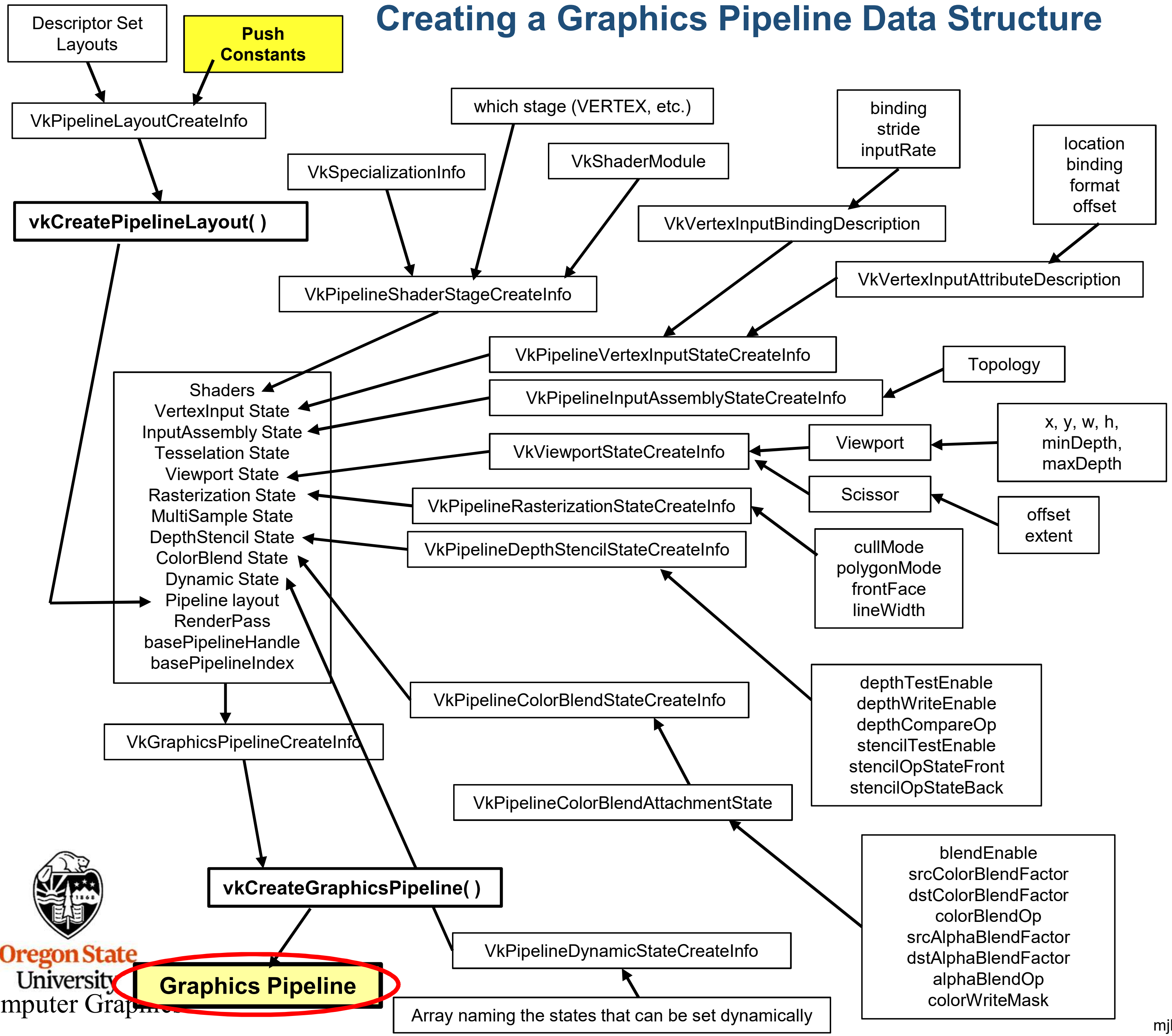
Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu

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 mat4 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.

Creating a Graphics Pipeline Data Structure



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:

```
layout( push_constant ) uniform matrix
{
        mat4 modelMatrix;
} Matrix;
```

On the application side, push constants are pushed at the shaders by giving 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_TESSELLATION_CONTROL_SHADER_BIT
VK_PIPELINE_STAGE_TESSELATION_EVALUATION_SHADER_BIT
VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT

size is in bytes

pValues is a void * pointer to the data, which, in this 4x4 matrix example, would be of type **glm::mat4**.

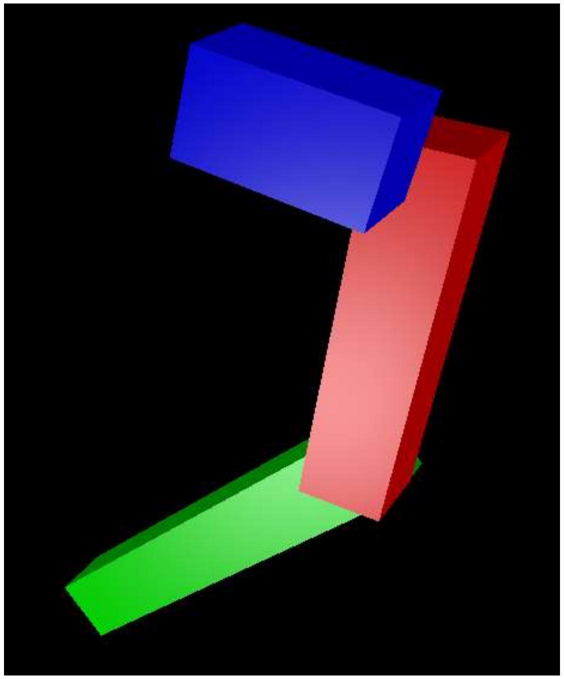
Setting up the Push Constants for the Graphics Pipeline Data Structure

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

```
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( glm::mat4 );

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 );
```

A Robotic Example using Push Constants

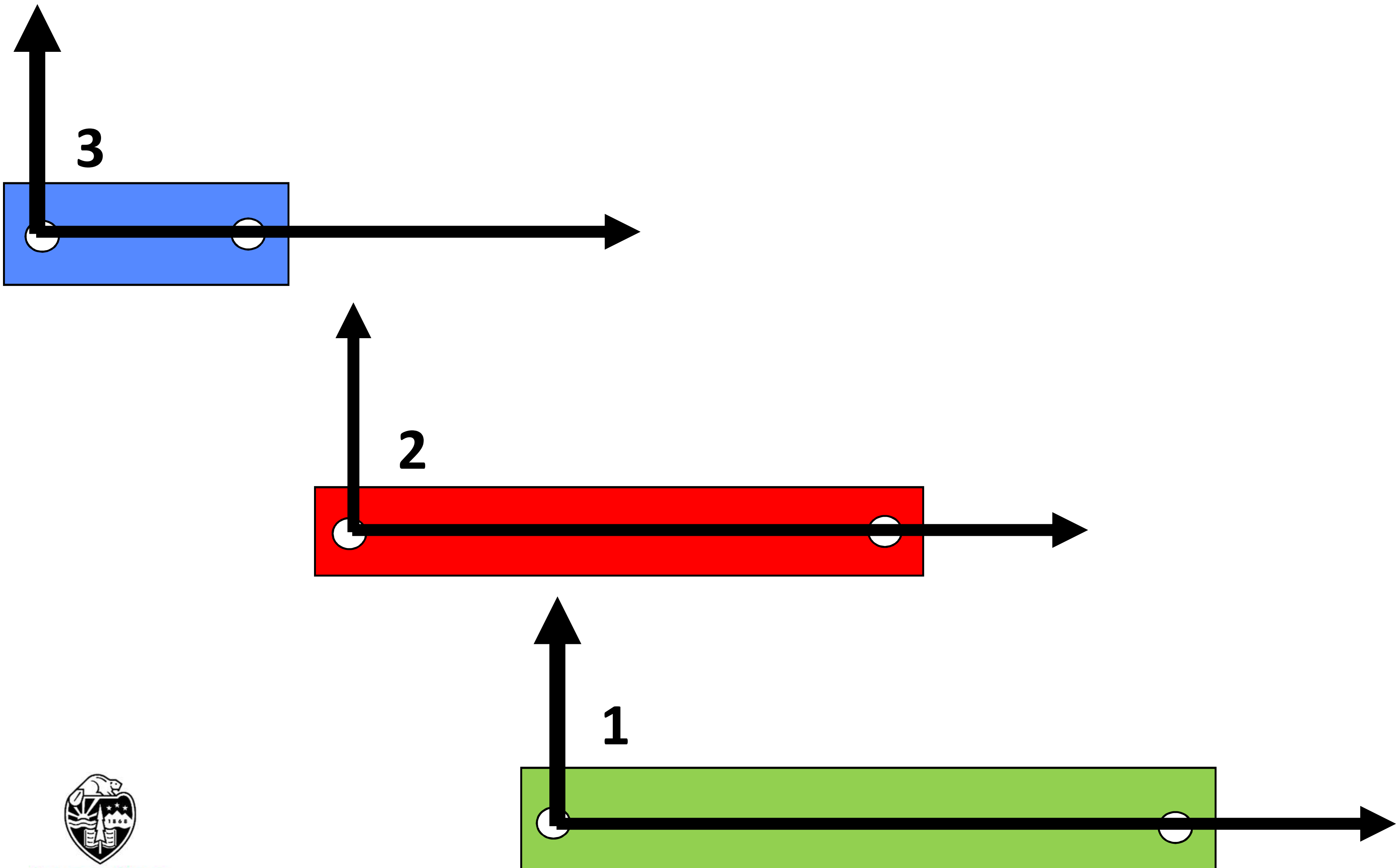
A robotic animation (i.e., a hierarchical transformation system)



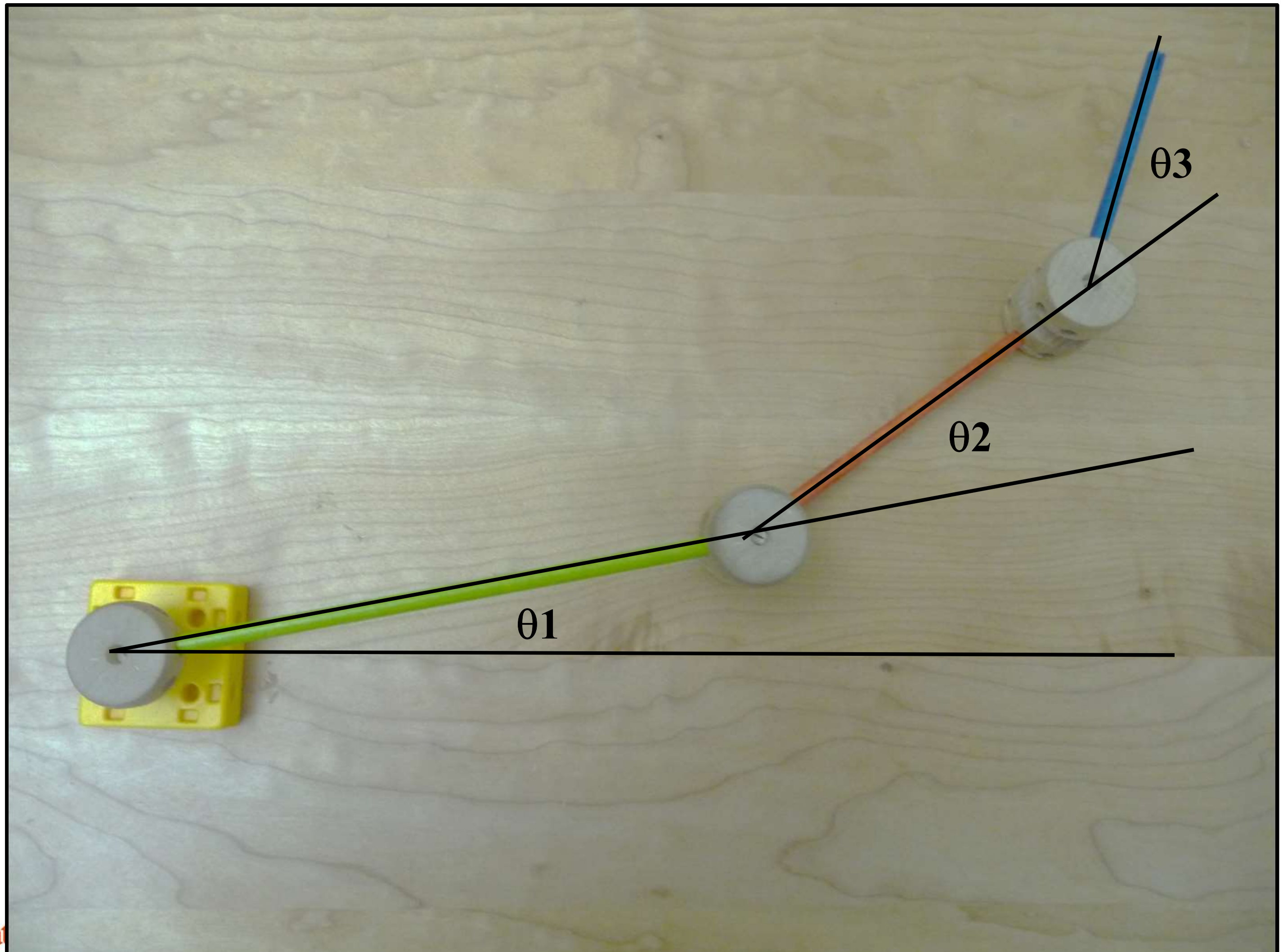
Where each arm is represented by:

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

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

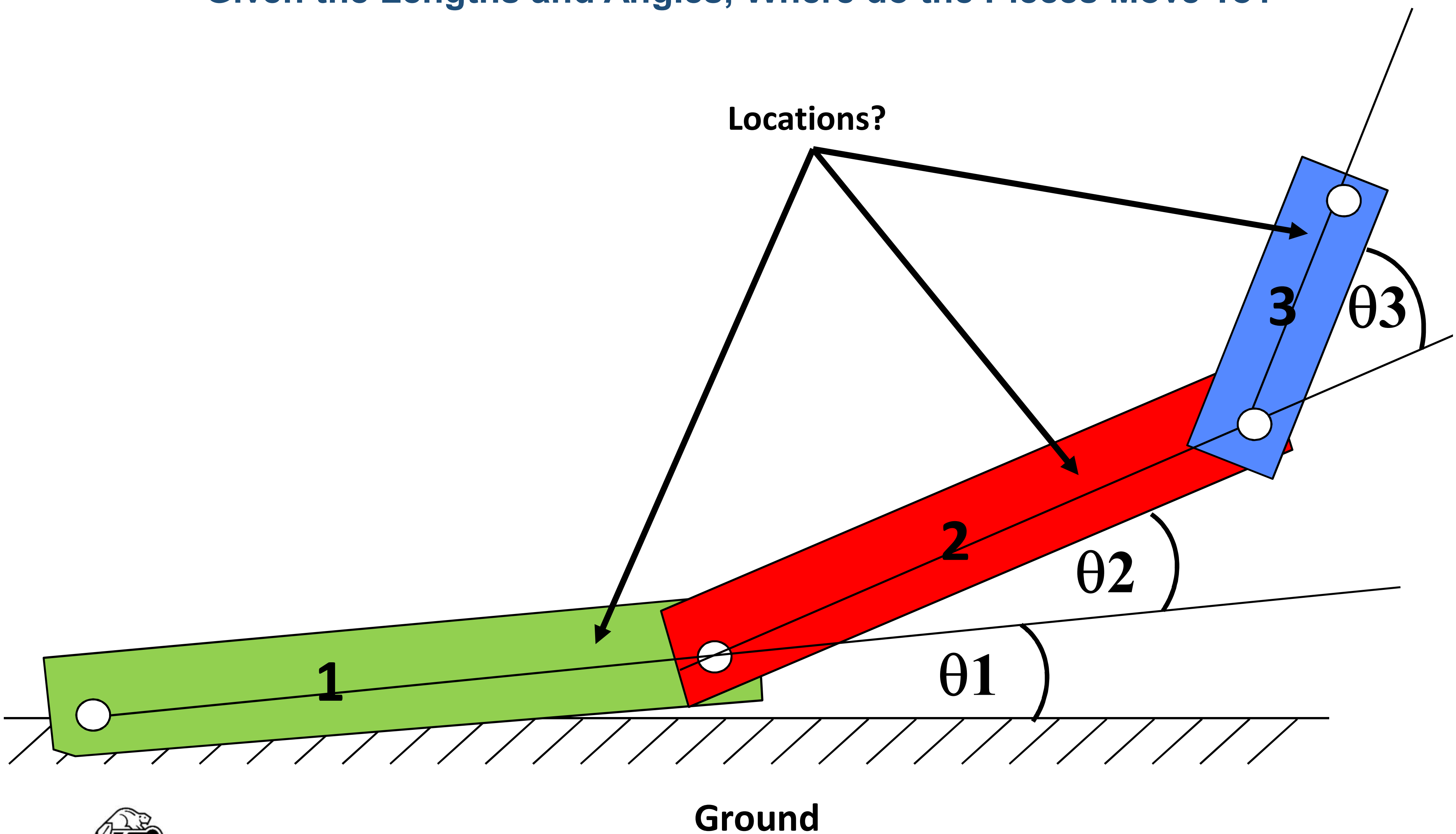
Forward Kinematics:
You Start with Separate Pieces, all Defined in their Own Local Coordinate System



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?



Positioning Part #1 With Respect to Ground

1. Rotate by Θ1
2. Translate by $T_{1/G}$

Code it →

$$\left[M_{1/G}\right]=\left[T_{1/G}\right]*\left[R_{\theta 1}\right]$$

← Say it

Why Do We Say it Right-to-Left?

Write it →

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

← Say it

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

$$\begin{Bmatrix} x' \\ y' \\ z' \\ 1 \end{Bmatrix} = \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{Bmatrix} x \\ y \\ z \\ 1 \end{Bmatrix}$$

$$\begin{Bmatrix} x' \\ y' \\ z' \\ 1 \end{Bmatrix} = [M_{1/G}] \begin{Bmatrix} x \\ y \\ z \\ 1 \end{Bmatrix} = [T_{1/G}] * [R_{\theta 1}] * \begin{Bmatrix} x \\ y \\ z \\ 1 \end{Bmatrix}$$

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 #2 With Respect to Ground

1. Rotate by Θ2
2. Translate the length of part 1
3. Rotate by Θ 1
4. Translate by $T_{1/G}$

Code it →

$$\left[M_{2/G}\right] = \left[T_{1/G}\right] * \left[R_{\theta 1}\right] * \left[T_{2/1}\right] * \left[R_{\theta 2}\right]$$

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

← Say it

Positioning Part #3 With Respect to Ground

1. Rotate by Θ3
2. Translate the length of part 2
3. Rotate by Θ2
4. Translate the length of part 1
5. Rotate by Θ1
6. Translate by $T_{1/G}$

Code it →

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

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

← Say it

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 );      // green
        Arm1.armScale  = 6.f;

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

        Arm3.armMatrix = glm::mat4( 1. );
        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


```
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 = 5;
        vplci.pSetLayouts = DescriptorSetLayouts;
        vplci.pushConstantRangeCount = 1;
        vplci.pPushConstantRanges = vpcr;

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

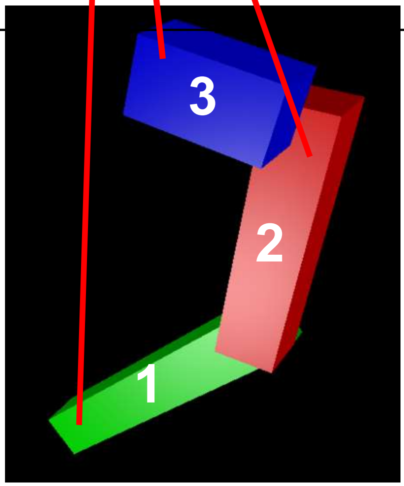
In the *UpdateScene( )* Function

```
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.);

glm::mat4 m1g = glm::mat4( 1. );    // identity
m1g = glm::translate(m1g, glm::vec3(0., 0., 0.));
m1g = glm::rotate(m1g, rot1, zaxis);                    // [T]*[R]

glm::mat4 m21 = glm::mat4( 1. );    // identity
m21 = glm::translate(m21, glm::vec3(2.*Arm1.armScale, 0., 0.));
m21 = glm::rotate(m21, rot2, zaxis);                    // [T]*[R]
m21 = glm::translate(m21, glm::vec3(0., 0., 2.));      // z-offset from previous arm

glm::mat4 m32 = glm::mat4( 1. );    // identity
m32 = glm::translate(m32, glm::vec3(2.*Arm2.armScale, 0., 0.));
m32 = glm::rotate(m32, rot3, zaxis);                    // [T]*[R]
m32 = glm::translate(m32, glm::vec3(0., 0., 2.));      // z-offset from previous arm

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

In the *RenderScene( )* Function


```
VkBuffer buffers[1]   =  { MyVertexDataBuffer.buffer };

vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, buffers, offsets );

vkCmdPushConstants( CommandBuffers[nextImageIndex], GraphicsPipelineLayout,
            VK_SHADER_STAGE_ALL, 0, sizeof(struct arm),  (void *)&Arm1 );
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );


vkCmdPushConstants( CommandBuffers[nextImageIndex], GraphicsPipelineLayout,
            VK_SHADER_STAGE_ALL, 0, sizeof(struct arm),  (void *)&Arm2 );
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );


vkCmdPushConstants( CommandBuffers[nextImageIndex], GraphicsPipelineLayout,
            VK_SHADER_STAGE_ALL, 0, sizeof(struct arm),  (void *)&Arm3 );
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
```

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


```
layout( push_constant ) uniform arm
{
	mat4  armMatrix;
	vec3  armColor;
	float   armScale;		// scale factor in x
} 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 = PVMM * vec4( bVertex, 1. );	// Projection * Viewing * Modeling matrices
```