Instancing – What and why?

- Instancing is the ability to draw the same object multiple times
- It uses all the same vertices and graphics pipeline each time
- It avoids the overhead of the program asking to have the object drawn again, letting the GPU/driver handle all of that

```c
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );
```

But, this will only get us multiple instances of identical objects drawn on top of each other. How can we make each instance look differently?
Making each Instance look differently -- Approach #1

Use the built-in vertex shader variable `gl_InstanceIndex` to define a unique display property, such as position or color.

`gl_InstanceIndex` starts at 0

In the vertex shader:

```cpp
int NUMINSTANCES = 16;
float DELTA = 3.0;

float xdelta = DELTA * float( gl_InstanceIndex % 4 );
float ydelta = DELTA * float( gl_InstanceIndex / 4 );
vColor = vec3( 1., float( (1.+gl_InstanceIndex) ) / float( NUMINSTANCES ), 0. );

xdelta -= DELTA * sqrt( float(NUMINSTANCES) ) / 2.;
ydelta -= DELTA * sqrt( float(NUMINSTANCES) ) / 2.;
vec4 vertex = vec4( aVertex.xyz + vec3( xdelta, ydelta, 0. ), 1. );

gl_Position = PVM * vertex;
```
Making each Instance look differently -- Approach #2

Put the unique characteristics in a uniform buffer and reference them

Still uses `gl_InstanceIndex`

In the vertex shader:

```glsl
layout( std140, set = 3, binding = 0 ) uniform colorBuf
{
    vec3 uColors[1024];
} Colors;

out vec3 vColor;

    ...

int index = gl_InstanceIndex % 1024;       // 0 - 1023
vColor = Colors.uColors[ index ];

gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```
Making each Instance look differently -- Approach #3

Put a series of unique characteristics in a data buffer, one element per instance.

Read a new characteristic for each instance

Internally uses `gl_InstanceIndex`, but you don’t
This is just the Vertex Input State Portion of the Graphics Pipeline Structure
<table>
<thead>
<tr>
<th>VkVertexInputBindingDescription</th>
<th>vvibd[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>// an array containing one of these per buffer being used</td>
</tr>
<tr>
<td>vvibd[0].binding = 0;</td>
<td>// which binding # this is</td>
</tr>
<tr>
<td>vvibd[0].stride = sizeof(struct vertex);</td>
<td>// bytes between successive</td>
</tr>
<tr>
<td>vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;</td>
<td></td>
</tr>
</tbody>
</table>

This definition says that we should advance through the input buffer by this much every time we hit a new `vertex`
How We Constructed the Graphics Pipeline Structure Before

```c
VkVertexInputAttributeDescription vviad[4];
    // an array containing one of these per vertex attribute in all bindings
    // 4 = vertex, normal, color, texture coord
vviad[0].location = 0;                  // location in the layout decoration
vviad[0].binding = 0;                   // which binding description this is part of
vviad[0].format = VK_FORMAT_VEC3;       // x, y, z
vviad[0].offset = offsetof( struct vertex, position );                  // 0

vviad[1].location = 1;
vviad[1].binding = 0;
vviad[1].format = VK_FORMAT_VEC3;       // nx, ny, nz
vviad[1].offset = offsetof( struct vertex, normal );                    // 12

vviad[2].location = 2;
vviad[2].binding = 0;
vviad[2].format = VK_FORMAT_VEC3;       // r, g, b
vviad[2].offset = offsetof( struct vertex, color );                     // 24

vviad[3].location = 3;
vviad[3].binding = 0;
vviad[3].format = VK_FORMAT_VEC2;       // s, t
vviad[3].offset = offsetof( struct vertex, texCoord );                  // 36
```
VkPipelineVertexInputStateCreateInfo \( vpvisci \);

// used to describe the input vertex attributes
vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
vpvisci.pNext = nullptr;
vpvisci.flags = 0;

\[ \text{vpvisci.vertexBindingDescriptionCount = 1;} \]
\[ \text{vpvisci.pVertexBindingDescriptions = vvibd;} \]

\[ \text{vpvisci.vertexAttributeDescriptionCount = 4;} \]
\[ \text{vpvisci.pVertexAttributeDescriptions = vviad;} \]

VkGraphicsPipelineCreateInfo \( vgpci \);

vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.flags = 0;

\[ \text{vgpci.pVertexInputState = \&vpvisci;} \]

result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN \&vgpci, PALLOCATOR, OUT pGraphicsPipeline );
Let’s assign a different color per instance.
Create a data buffer with one glm::vec3 (to hold r, g, b) for each instance.

```c
VkVertexInputBindingDescription vvibd[2];
vvibd[0].binding = 0; // which binding # this is
vvibd[0].stride = sizeof( struct vertex ); // bytes between successive
vvibd[0].inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

vvibd[1].binding = 1; // which binding # this is
vvibd[1].stride = sizeof( glm::vec3 ); // bytes between successive entries
vvibd[1].inputRate = VK_VERTEX_INPUT_RATE_INSTANCE;
```

This definition says that we should advance through the input buffer by this much every time we hit a new instance.
How We Construct the Graphics Pipeline Structure Now

Let’s assign a different color per Instance.
Create a data buffer with one glm::vec3 (to hold r, g, b) for each Instance.

```cpp
VkVertexInputAttributeDescription vviad[5];
    // an array containing one of these per vertex attribute in all bindings
    // 4 = vertex, normal, color, texture coord
vviad[0].location = 0;       // location in the layout decoration
vviad[0].binding = 0;        // which binding description this is part of
vviad[0].format = VK_FORMAT_VEC3;    // x, y, z
vviad[0].offset = offsetof( struct vertex, position );       // 0

...  

vviad[5].location = 0;       // location in the layout decoration
vviad[5].binding = 1;        // which binding description this is part of
vviad[5].format = VK_FORMAT_VEC3;    // r, g, b
vviad[5].offset = 0;         // just one element, so offset is 0
```
How We Construct the Graphics Pipeline Structure Now

Let’s assign a different color per Instance.
Create a data buffer with one glm::vec3 (to hold r, g, b) for each Instance.

```cpp
VkPipelineVertexInputStateCreateInfo vpvisci;
vpvisci.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
vpvisci.pNext = nullptr;
vpvisci.flags = 0;

vpvisci.vertexBindingDescriptionCount = 2;
vpvisci.pVertexBindingDescriptions = vvibd;

vpvisci.vertexAttributeDescriptionCount = 5;
vpvisci.pVertexAttributeDescriptions = vviad;

Note: same names as before, but different sizes
```

```cpp
VkGraphicsPipelineCreateInfo vgpci;
vgpci.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
vgpci.pNext = nullptr;
vgpci.flags = 0;

...

vgpci.pVertexInputState = &vpvisci;
...

result = vkCreateGraphicsPipelines( LogicalDevice, VK_NULL_HANDLE, 1, IN &vgpci,
PALLOCATOR, OUT pGraphicsPipeline );
```
#version 400
#extension GL_ARB_separate_shader_objects : enable
#extension GL_ARB_shading_language_420pack : enable

... 

layout( location = 0 ) in vec3 aVertex;
layout( location = 1 ) in vec3 aNormal;
layout( location = 2 ) in vec3 aColor;
layout( location = 3 ) in vec2 aTexCoord;

layout( location = 4 ) in vec3 aInstanceColor;

layout( location = 0 ) out vec3 vNormal;
layout( location = 1 ) out vec3 vColor;
layout( location = 2 ) out vec2 vTexCoord;

void main( )
{
    mat4 PVM = Matrices.uProjectionMatrix * Matrices.uViewMatrix * Matrices.uModelMatrix;

    vNormal = normalize( vec3( Matrices.uNormalMatrix * vec4(aNormal, 1.) ) );
    //vColor = aColor;
    vColor = aInstanceColor;
    vTexCoord = aTexCoord;

    gl_Position = PVM * vec4( aVertex, 1. );
}