A Better Way

This is fine for a 3-vertex example. But imagine a scene involving millions of vertices (and no, that's not an exaggeration for high-end applications). Moving objects this way means having to copy millions of vertices from the original vertex data, add an offset to each of them, and then upload that data to an OpenGL buffer object. And all of that is *before* rendering. Clearly there must be a better way; games can not possibly do this every frame and still hold decent framerates.

Actually for quite some time, they did. In the pre-GeForce 256 days, that was how all games worked. Graphics hardware just took a list of vertices in clip space and rasterized them into fragments and pixels. Granted, in those days, we were talking about maybe 10,000 triangles per frame. And while CPUs have come a long way since then, they have not scaled with the complexity of graphics scenes.

The GeForce 256 (note: not a GT 2xx card, but the very first GeForce card) was the first graphics card that actually did some from of vertex processing. It could store vertices in GPU memory, read them, do some kind of transformation on them, and then send them through the rest of the pipeline. The kinds of transformations that the old GeForce 256 could do were quite useful, but fairly simple.

Having the benefit of modern hardware and OpenGL 3.x, we have something far more flexible: vertex shaders.

Remember what it is that we are doing. We compute an offset. Then we apply that offset to each vertex position. Vertex shaders are given each vertex position. So it makes sense to simply give the vertex shader the offset and let it compute the final vertex position, since it operates on each vertex. This is what `vertPositionOffset.cpp` does.

The vertex shader is found in `data\positionOffset.vert`. The vertex shader used here is as follows:

Example 3.4. Offsetting Vertex Shader

```
#version 330

layout(location = 0) in vec4 position;
uniform vec2 offset;

void main()
{
    vec4 totalOffset = vec4(offset.x, offset.y, 0.0, 0
    gl_Position = position + totalOffset;
}
```

After defining the input `position`, the shader defines a 2-dimensional vector `offset`. But it defines it with the term `uniform`, rather than `in` or `out`. This has a particular meaning.

Shaders and Granularity. Recall that, with each execution of a shader, the shader gets new values for variables defined as `in`. Each time a vertex shader is called, it gets a different set of inputs from the vertex attribute arrays and buffers. That is useful for vertex position data, but it is not what we want for the offset. We want each vertex to use the *same* offset; a "uniform" offset, if you will.

Variables defined as `uniform` do not change at the same frequency as variables defined as `in`. Input variables change with every execution of the shader. Uniform variables (called *uniforms*) change only between executions of rendering calls. And even then, they only change when the user sets them explicitly to a new value.

Vertex shader inputs come from vertex attribute array definitions and buffer objects. By contrast, uniforms are set directly on program objects.

In order to set a uniform in a program, we need two things. The first is a uniform location. Much like with attributes, there is an index that refers to a specific uniform value. Unlike attributes, you cannot set this location yourself; you must query it. In this tutorial, this is done in the `InitializeProgram` function, with this line:

```
offsetLocation = glGetUniformLocation(theProgram, "off
```

The function `glGetUniformLocation` retrieves the uniform location for the uniform named by the second parameter. Note that just because a uniform is defined in a shader, GLSL does not *have* to provide a location for it. It will only have a location if the uniform is actually used in the program, as we see in the vertex shader; `glGetUniformLocation` will return -1 if the uniform has no location.

Once we have the uniform location, we can set the uniform's value. However, unlike retrieving the uniform location, setting a uniform's value requires that the program be currently in use with `glUseProgram`. Thus, the rendering code looks like this:

Example 3.5. Draw with Calculated Offsets

```
void display()
{
    float fXOffset = 0.0f, fYOffset = 0.0f;
    ComputePositionOffsets(fXOffset, fYOffset);

    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glUseProgram(theProgram);

    glUniform2f(offsetLocation, fXOffset, fYOffset);
```

```
    glBindBuffer(GL_ARRAY_BUFFER, positionBufferObje
```

We use `ComputePositionOffsets` to get the offsets, and then use `glUniform2f` to set the uniform's value. The buffer object's data is never changed; the shader simply does the hard work. Which is why those shader stages exist in the first place.

Standard OpenGL Nomenclature

The function `glUniform2f` uses a special bit of OpenGL nomenclature. Specifically, the “2f” part.

Uniforms can have different types. And because OpenGL is defined in terms of C rather than C++, there is no concept of function overloading. So functions that do the same thing which take different types must have different names. OpenGL has standardized its naming convention for these.

Functions like `glUniform` have a suffix. The first part of the suffix is the number of values it takes. In the above case, `glUniform2f` takes 2 values, in addition to the regular parameters. The basic `glUniform` parameters are just the uniform location, so `glUniform2f` takes a uniform location and two values.

The type of values it takes is defined by the second part. The possible type names and their associated types are:

b	signed byte	ub	unsigned byte
s	signed short	us	unsigned short
i	signed int	ui	unisgned int
f	float	d	double

Note that OpenGL has special typedefs for all of these types. GLfloat

Vector Math. You may be curious about how these lines work:

```
vec4 totalOffset = vec4(offset.x, offset.y, 0.0, 0.0);
gl_Position = position + totalOffset;
```

The vec4 that looks like a function here is a constructor; it creates a vec4 from 4 floats. This is done to make the addition easier.

The addition of `position` to `totalOffset` is a component-wise addition. It is a shorthand way of doing this:

```
gl_Position.x = position.x + totalOffset.x;
gl_Position.y = position.y + totalOffset.y;
gl_Position.z = position.z + totalOffset.z;
gl_Position.w = position.w + totalOffset.w;
```

GLSL has a lot of vector math built in. The math operations are all component-wise when applied to vectors. However, it is illegal to add vectors of different dimensions to each other. So you cannot have a vec2 + vec4. That is why we had to convert `offset` to a vec4 before performing the addition.
