Programmable Vertex Pulling 21

Daniel Rákos

21.1 Introduction

OpenGL and today's GPUs provide a high degree of flexibility for acquiring geometry-related information from auxiliary buffers using the shader built-in constants provided by GLSL based on data granularity. The name `gl_VertexID` provides the index of the currently processed vertex, `gl_PrimitiveID`, which provides the index of the currently processed geometric primitive, and `gl_InstanceID` provides the index of the currently processed instance of an instanced draw command. Still, there are restrictions on how object information can be passed to the graphics pipeline if we use traditional methods for specifying geometric information using attribute arrays and an optional element array.

This chapter explores the possibility of taking advantage of some of the latest GPU technologies to provide a method that enables completely programmable *vertex pulling*, i.e., a programmable approach to fetch vertex attributes.

The possibility of implementing programmable vertex pulling has been available in OpenGL and in hardware for some time now, but this technique is rarely used in practice. The main reason behind this is that developers assume that fixed-functionality vertex pulling uses dedicated hardware to execute this task and thus can provide better performance.

However, OpenGL 3.x–capable hardware's unified architecture shows that fixed-function vertex pulling has to go through the very same hardware path that programmable buffer fetching does, including the cache hierarchy that is shared among all fetching units, including attribute, buffer, and texture fetches.

The main goal of this chapter is to implement a simple programmable vertex pulling shader with a sample vertex attribute setup and compare its performance with a setup that uses fixed-function vertex attribute fetching to demonstrate the performance characteristics of programmable vertex pulling.

Further, I will present common use cases where programmable vertex pulling can provide additional flexibility and/or performance over the traditional approaches.

21.2 Implementation

The core of the implementation of programmable vertex pulling is built around the functionality provided by *buffer textures* [Brown 08]. These textures provide a method that allows every shader stage to fetch arbitrary data from buffer objects. Trivially, this functionality alone is enough to implement programmable vertex pulling, as the only change that has to be made compared to fixed-function vertex pulling is that all vertex attributes and, optionally, the indices are manually fetched in the vertex shader.

I will distinguish two types of programmable vertex pulling methods:

1. Programmable attribute fetching. In this case, we will still use fixed-function indexed primitive rendering, but the vertex attributes will be manually fetched in the vertex shader.
2. Fully programmable vertex pulling. Vertex indexing will be done in the vertex shader together with the vertex attribute fetching.

In our sample implementation, we use a simple vertex attribute setup of 32 bytes/vertex (3 floats for position, 3 floats for normal, and 2 floats for texture coordinates) all stored in an interleaved buffer format. In the case of fixed-function vertex pulling, we use an element array for *indexed primitive rendering*. In the case of our programmable vertex pulling implementation, the element array will be fed to the vertex shader as the only vertex attribute array, and we will implement indexed primitive rendering programmatically in the vertex shader (i.e., fully programmable vertex pulling).

The vertex shader used for both fixed-function and programmable vertex pulling is presented in Listing 21.1, where the preprocessor directive `PROGRAMMABLE` is defined only in the case of the latter. The shader transforms the vertex position into clip space, the normal to view space, and passes them together with the texture coordinate set to subsequent stages of the rendering pipeline.

Obviously, the client-side code setup for the vertex arrays and the buffer textures is different in both cases. This is presented in Listing 21.2 using the same preprocessor directive for selecting between the two rendering paths.

Now, in this example, there is no additional flexibility provided by programmable vertex pulling; however, if the information required by the vertex shader is not

```
#version 420 core

layout(std140, binding = 0) uniform transform {
    mat3 NormalMatrix;
    mat4 MVPMatrix;
} Transform;

#ifdef PROGRAMMABLE
layout(location = 0) in int inIndex;
layout(binding = 0) uniform samplerBuffer attribBuffer;
#else
layout(location = 0) in vec3 inVertexPosition;
layout(location = 1) in vec3 inVertexNormal;
layout(location = 2) in vec2 inVertexTexCoord;
#endif

layout out vec3 outVertexNormal;
layout out vec2 outVertexTexCoord;

out gl_PerVertex {
    vec4 gl_Position;
};

void main(void) {
#ifdef PROGRAMMABLE
    vec4 attrib0 = texelFetch(attribBuffer, inIndex * 2);
    vec4 attrib1 = texelFetch(attribBuffer, inIndex * 2 + 1);
    vec3 inVertexPosition = attrib0.xyz;
    vec3 inVertexNormal = vec3(attrib0.w, attrib1.xy);
    vec2 inVertexTexCoord = attrib1.zw;
#endif
    gl_Position = MVPMatrix * vec4(inVertexPosition, 1.f);
    outVertexNormal = NormalMatrix * inVertexNormal;
    outVertexTexCoord = inVertexTexCoord;
}
```

Listing 21.1. Sample vertex shader that can perform fixed-function or programmable vertex pulling.

```
#ifdef PROGRAMMABLE
/* setup index buffer as the only vertex attribute */
    glBindBuffer(GL_ARRAY_BUFFER, indexBuffer);
    glEnableVertexAttribArray(0);
    glVertexAttribIPointer(0, 1, GL_INT, 4, NULL);
/* configure the buffer texture to use the vertex attribute buffer as storage */
    glBindTexture(GL_TEXTURE_BUFFER, bufferTexture);
    glTexBuffer(GL_TEXTURE_BUFFER, GL_RGBA32F, vertexBuffer);
#else
/* use the index buffer as element array buffer */
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
/* use the vertex buffer to set up interleaved vertex attributes */
    glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 32, 0);
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 32, 12);
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 32, 24);
#endif
```

Listing 21.2. Client-side configuration of the vertex attributes for fixed-function and programmable vertex pulling.

different for every vertex, programmable vertex pulling can be advantageous. Fixed-function vertex pulling provides a mechanism via instanced arrays [Helferty et al. 08] that makes it possible to fetch some vertex attributes with a smaller frequency than every vertex, but it is limited to the possibility of fetching new attributes at every *n*th instance. With programmable vertex pulling, this frequency can be arbitrary.

One can use the built-in shader variables `gl_VertexID`, `gl_PrimitiveID`, and `gl_InstanceID` to control the rate of attribute fetches, but on OpenGL 4.x–capable hardware, even nonuniform fetching frequencies can be achieved by using custom shader logic and *atomic counters* [Licea-Kane et al. 11].

21.3 Performance

One of the preconceptions that scares off developers from using programmable vertex pulling where traditional fixed-function vertex pulling is simply not flexible enough to solve a particular problem is performance. In order to falsify this preconception, I've executed some tests that reveal the strengths and weaknesses and the relative performance of programmable vertex pulling compared to the fixed functionality. The tests were executed on OpenGL 3– and OpenGL 4–capable GPUs from different vendors and use the Stanford Dragon and Buddha models consisting of 871,414 and 1,087,716 indexed triangles, respectively.

The tests use the vertex shader and client-side setup presented in Listing 21.1 and Listing 21.2. In the case of fixed-function vertex pulling and programmable attribute fetching, we render the models using a single `glDrawElements` call, while in the case of fully programmable vertex pulling, the `glDrawArrays` command is used because, in this case, we don't intend to utilize fixed-function indexed primitive rendering.

The measurements have been done so that we can eliminate all the fragment processing overhead by rendering the models outside the view frustum, as we are primarily aiming to measure vertex-processing cost differences between the three techniques. We use timer queries [Daniell 10] to measure the GPU time required to render the models (see Figure 21.1). Unfortunately, based on our tests, timer queries return very dissimilar values in the case of the two hardware vendors, which may be either the result of driver implementation or hardware architecture differences. However, as we are only interested in the relative performance of the three vertex pulling techniques, this does not really affect us.

Table 21.1 shows that programmable attribute fetching is as fast as fixed-function vertex pulling on modern GPUs, even though we may have expected a slight performance penalty because of the possible additional latency incurred by performing the vertex attribute fetching inside the vertex shader. This shows us that the sophisticated latency-hiding mechanisms of current GPUs eliminate this cost.

The picture is a bit less bright when using fully programmable vertex pulling, though the Radeons provide acceptable performance there as well. The reason behind

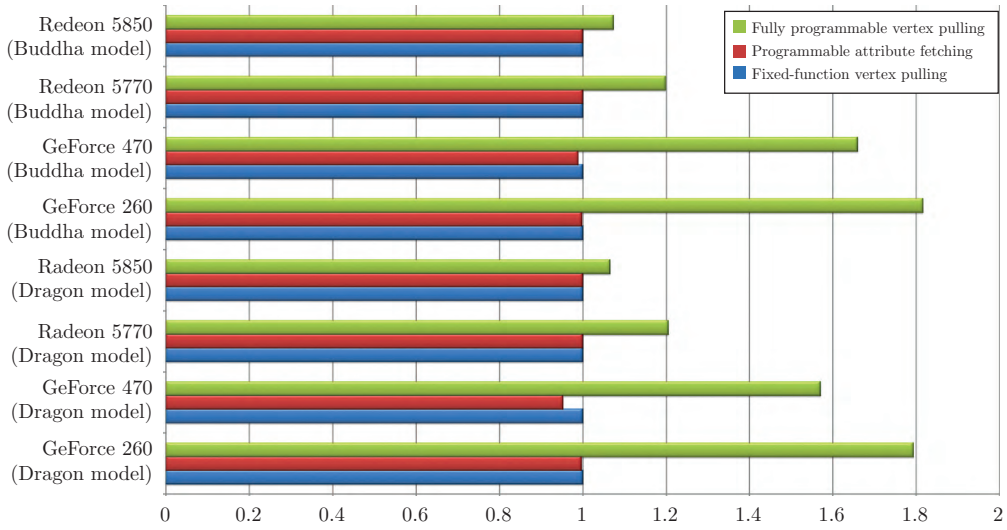

Figure 21.1. Relative GPU time of rendering the Stanford Dragon and Buddha models on various GPUs using programmable vertex pulling compared to fixed-function vertex pulling (lower values are better).

the performance penalty for fully programmable vertex pulling is the fact that it cannot take advantage of the *post-transform cache*, which can greatly increase the speed of indexed primitive rendering. I have to also mention that the models were not optimized for maximum post-transform cache usage, so the time difference may be even higher in real-life scenarios. The advantage of the Radeon GPUs when

GPU	Model	Fixed-function vertex pulling	Programmable attribute fetching		Fully programmable vertex pulling	
GPU time		**Absolute**	**Absolute**	**Relative**	**Absolute**	**Relative**
GeForce 260	Dragon	3.291 ms	3.281 ms	-0.3 %	5.902 ms	+79.3 %
(GL3)	Buddha	4.056 ms	4.047 ms	-0.2 %	7.366 ms	+81.6 %
GeForce 470	Dragon	0.786 ms	0.748 ms	-4.8 %	1.234 ms	+57.0 %
(GL4)	Buddha	0.928 ms	0.918 ms	-1.1 %	1.540 ms	+65.9 %
Radeon 5770	Dragon	10.287 ms	10.288 ms	0.0 %	12.393 ms	+20.5 %
(GL4)	Buddha	13.377 ms	13.381 ms	0.0 %	16.034 ms	+19.9 %
Radeon 5850	Dragon	8.896 ms	8.897 ms	0.0 %	9.471 ms	+6.5 %
(GL4)	Buddha	11.177 ms	11.177 ms	0.0 %	12.009 ms	+7.4 %

Table 21.1. Absolute and relative GPU time for rendering the Stanford Dragon and Buddha models on various GPUs using programmable vertex pulling compared to fixed-function vertex pulling (lower values are better).

using fully programmable vertex pulling makes me believe that AMD GPUs are less dependent on efficient post-transform cache usage.

Based on the results, my verdict is that programmable vertex pulling has no overhead compared to fixed-function vertex pulling in the following cases:

- When rendering nonindexed primitives (e.g., triangle strips).
- When rendering indexed primitives using fixed-function index handling.

While programmable index handling is also an option, when the meshes rely heavily on the usage of the post-transform cache, the performance of fully programmable vertex pulling can be prohibitive.

21.4 Application

We've already mentioned that programmable vertex pulling can enable us to control the frequency of vertex attribute consumption. This means that we can, for example, pass normals on a per-triangle basis instead of per-vertex; in a similar fashion, we can select a single layer of a texture array, again, on a per-primitive basis. Additionally, there might be attributes that have to be consumed from the attribute buffer only in certain cases, in which case, we can use an atomic counter to supervise our current position in the buffer. This can decrease memory size and bandwidth requirements of storing and fetching attributes, thus resulting in better overall performance.

Programmable vertex pulling can handle interleaved and separate data buffers as well, but it also makes possible the use of arbitrary data structures to store the vertex attributes or other information that may be needed by the vertex shader. This can include even multiple indirections, though performance may be a concern when doing an excessive number of buffer lookup indirections. This also enables the possibility of handling multiple vertex formats in a single vertex shader and thus can reduce the number of vertex format setups at the cost of a fairly low runtime overhead in most cases. While at the time of this, writing OpenGL does not support structure fetches, the introduction of POD (plain old data structure) fetches can further simplify the use of programmable vertex pulling in these cases.

Another application of programmable vertex pulling can be attribute-less rendering. This can come handy when rendering simple primitives like full-screen triangles for postprocessing or simple light-volume primitive rendering in the case of deferred rendering methods, but it can also be used to dynamically generate parametric curves and surfaces in the vertex shader. Neither of these require any vertex attribute arrays, as all of them can be implemented using a few uniform variables as parameters. Considering that ALU capacity is usually higher than memory bandwidth, programmable vertex pulling can greatly increase the rendering performance in these situations.

Besides performance-critical applications, where certain types of data structures would simply not be feasible, CAD software can benefit from programmable vertex pulling. CAD software, depending on the target domain, uses various data structures for storing the topology of the mesh internally. These internal representations can be based on, e.g., the winged edge model [Baumgart 75], quad-edge data structure [Guibas and Stolfi 85], combinatorial maps or boundary representation models. As CAD software, in general, uses fixed-function vertex pulling, it usually has to maintain two copies of the data, one for internal usage and one for rendering. Taking advantage of programmable vertex pulling can potentially eliminate the need for the second copy by enabling the rendering pipeline to parse and display the mesh data in its original form, used internally by the CAD software.

21.5 Limitations

The biggest issue with fully programmable vertex pulling is that we cannot take advantage of the post-transform vertex cache that otherwise greatly increases the speed of indexed primitive rendering. In order to take advantage of this optimization, we require new hardware and APIs to be able to explicitly tag vertices emitted by a vertex shader invocation.

In fixed-function vertex pulling, this is done up front by tagging the vertices with their indices. My proposal is to introduce a new output parameter called `gl_VertexTag` to the vertex shader language that would be used by the post-transform vertex cache to tag the received vertices. While this approach would still not allow us to discard vertices that are already processed on a wavefront, in practice, it could potentially increase the performance of fully programmable vertex pulling to as close as possible to the speed of its fixed-function counterpart. This feature would also allow the post-transform cache to function efficiently in other cases where the system disables it because the vertex shader can have side effects, as in the case where the vertex shader uses atomic counters or load/store images [Bolz et al. 11].

Another approach to optimize programmable vertex pulling in the case of programmatically indexed primitives is to implement a sort of post-transform vertex cache in the vertex shader. This option is actually possible using OpenGL 4.2 by taking advantage of atomic counters and load/store images to store already processed vertices, although the performance of such an approach may still be much lower than that of the fixed-function post-transform vertex cache.

A further limitation of programmable vertex pulling is that manual format conversion may be needed in the vertex shader when we would like to use interleaved attribute arrays that contain attributes with multiple different data formats. For such cases, our proposal is to attach the same buffer object to multiple buffer textures using different internal formats and access the appropriate buffer texture in the vertex shader that is as close to the target format as possible to minimize the ALU cost required to convert the values to the intended representation.

21.6 Conclusion

Based on the performance results, I can say that programmable attribute fetching is a viable alternative to fixed-function vertex pulling even in the case when the fixed-function method could be applied as well, considering that there is no latency incurred by manual attribute fetching on most GPUs. However, the strength of programmable vertex pulling appears when storing the attributes in a structure that is suitable for traditional rendering is simply not feasible due to the size of the data set or in cases in which we already have an internal representation that we would like to work with that does not map well to any of the fixed-function attribute-specification methods.

As we've seen, it is pretty straightforward to implement programmable vertex pulling using the existing tool set provided by OpenGL, and the required hardware is only an OpenGL 3.x–capable GPU, though a much greater level of flexibility is available on OpenGL 4.x–capable GPUs. Actually, in theory, even earlier GPUs can take advantage of this technique if they support vertex texture fetches by storing the vertex attributes in a traditional 1D texture.

I've also shown that programmable vertex pulling can only be prohibitive from a performance point of view if we are using programmable indexed primitive rendering, as in this case the lack of post-transform vertex cache utilization can dramatically decrease the performance. I also proposed a few possible solutions to circumvent this issue.

Finally, I also discussed a few potential applications of the presented technique, both regarding interactive rendering and CAD software, and I also discussed the key limitations of programmable vertex pulling compared to its fixed-function counterpart.

There is need for a much longer and more in-depth study in order to be able to get a better picture of the capabilities and weaknesses of programmable vertex pulling, although I hope that this brief preview of the technique's potential captures the attention of readers to seek and find their own best use cases for it.

Bibliography

[Baumgart 75] Bruce G. Baumgart. "Winged-Edge Polyhedron Representation for Computer Vision." National Computer Conference, 1975.

[Bolz et al. 11] Jeff Bolz, Pat Brown, Barthold Lichtenbelt, Bill Licea-Kane, Eric Werness, Graham Sellers, Greg Roth, Nick Haemel, Pierre Boudier, and Piers Daniell. "ARB_shader_image_load_store." OpenGL extension specification, 2011.

[Brown 08] Pat Brown. "ARB_texture_buffer_object." OpenGL extension specification, 2008.

[Daniell 10] Piers Daniell. "ARB_timer_query." OpenGL extension specification, 2010.

[Guibas and Stolfi 85] Leonidas J. Guibas and Jorge Stolfi. "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams." *ACM Transactions on Graphics*, New York: ACM Press, 1985.

[Helferty et al. 08] James Helferty, Daniel Koch, Michael Gold, and John Rosasco. "ARB_instanced_arrays." OpenGL extension specification, 2008.

[Licea-Kane et al. 11] Bill Licea-Kane, Barthold Lichtenbelt, Chris Dodd, Eric Werness, Graham Sellers, Greg Roth, Jeff Bolz, Nick Haemel, Pat Brown, Pierre Boudier, and Piers Daniell. "ARB_shader_atomic_counters." OpenGL extension specification, 2011.