Attributed Vertex Clouds

Willy Scheibel, Stefan Buschmann,
Matthias Trapp, and Jürgen Döllner

1.1 Introduction

In todays computer graphics applications, large 3D scenes are rendered which consist of *polygonal geometries* such as triangle meshes. Using state-of-the-art techniques, this geometry is often represented on the GPU using vertex and index buffers, as well as additional auxiliary data such as textures or uniform buffers [Riccio and Lilley 13]. It is usually loaded or generated on the CPU and transferred to the GPU for efficient rendering using the programmable rendering pipeline. For polygonal meshes of arbitrary complexity, the described approach is indispensable. However, there are several types of simpler geometries (e.g., cuboids, spheres, tubes, or splats) that can be generated procedurally. For scenes that consist of a large numbers of such geometries, which are parameterized individually and potentially need to be updated regularly, another approach for representing and rendering these geometries may be beneficial.

In the following, we present an efficient data representation and rendering concept for such geometries, denoted as *attributed vertex clouds* (AVCs). Using this approach, geometry is generated on the GPU during execution of the programmable rendering pipeline. Instead of the actual geometry, an AVC contains a set of vertices which describe the target geometry. Each vertex is used as the argument for a function that procedurally generates the target geometry. This function is called a *transfer function*, and it is implemented using shader programs and therefore executed as part of the rendering process.

This approach allows for compact geometry representation and results in reduced memory footprints in comparison to traditional representations. Also, AVCs can be rendered using a single draw call, improving rendering performance. By shifting geometry generation to the GPU, the resulting *volatile* geometry can be controlled flexibly, i.e., its position, parameterization, and even the type of geometry can be modified without requiring state changes or uploading new data to the GPU. Furthermore, the concept is easy to implement and integrate into existing rendering systems, either as part of or in addition to other rendering approaches.

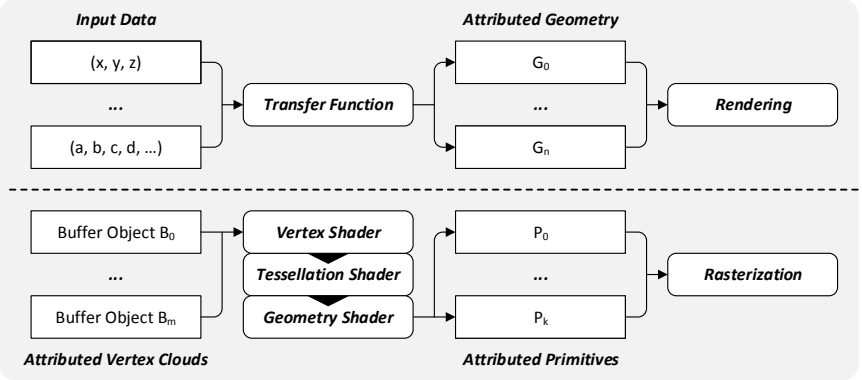


Figure 1.1. Concept overview of attributed vertex clouds. Data points with arbitrary attributes are passed to a transfer function that creates attributed geometry (attributed vertices arranged in primitives). More specific, one data point of an attributed vertex cloud is passed through the programmable rendering pipeline and transformed into attributed vertices and primitives.

1.2 Concept

An *attributed vertex cloud* consists of a number of vertices, each of which describes one instance of the target geometry. A single vertex can thereby contain either a parameterized description of the geometry [Overvoorde 14] (e.g., a cuboid can be specified by center point, extent, and orientation), or more generally an arbitrary data point, which by some function can be transformed into the desired geometry parameterization. For each vertex, a *transfer function* is applied, that converts the vertex into a parameterized description of the target geometry. This function determines the type and configuration of the target geometry based mainly on the input vertex, but it can also take into account other parameters such as global configuration, camera position, or interaction states. Finally, a geometry generation function is invoked with the selected parameters, creating the actual geometry [Kemen 12] which is then rasterized and rendered to the screen.
The AVC concept consists of the following parts:

Attribute Vertex Cloud. The AVC is a vertex buffer that contains a set of data points which are rendered as the geometry of a scene. Each vertex thereby represents one instance of the target geometry and provides all relevant attributes needed to select and configure the target geometry. However, the actual definition of one vertex may vary widely depending on the use case. A vertex can already contain a detailed configuration of the

geometry (e.g., it may describe a cube by its center position, extents, orientation, and color). In other use cases, a vertex may contain a data point of the target domain (e.g., a tracking point in a movement trajectory, consisting of the current geo-location, speed, and acceleration) or compressed attributes [Purnomo et al. 05]. The AVC is represented on the GPU as a vertex array buffer, consisting of several vertex attributes. It may also reference other buffers by index. This can be used to represent shared data between vertices, such as *per-group*, *per-batch*, or *global* attributes.

Transfer Function. The transfer function selects the configuration of the output geometry for each vertex. It is usually implemented in either the vertex or the geometry shader, taking a data point of the AVC as input and creating a geometry configuration vector as its output. The transfer function can depend not only on the values of the vertex itself, but other parameters can be used as well to influence the target geometry. For example, the current camera position can be used to determine the distance from the camera to the geometry that is about to be rendered. Based on that distance, the type and style of the generated geometry can be altered. This enables the implementation of level-of-detail and, to a certain degree, level-of-abstraction techniques. Further, interaction and navigation states (e.g., interactive filtering or selection by the user) can be applied to parameterize geometry generation. Also, global configuration parameters specified by the user can be taken into account, allowing the user to control or parameterize the generated geometry (e.g., select geometry types, provide size factors or constraints, etc.).

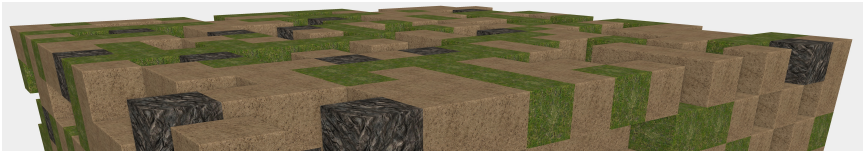
Geometry Generation. In the last step, the actual geometry is generated according to the configuration previously selected by the transfer function. This is usually implemented using tessellation and geometry shaders. Due to practical considerations, the implementations of transfer function and geometry generation can sometimes be merged into one. Depending on the use case, different types of geometries can be emitted for each individual vertex. Where applicable, vertices may also be culled [Rákos 10]. If such dynamic geometry generation is used, we suggest to separate the draw calls to one geometry type per batch for optimized performance.

1.3 Applications for Attributed Vertex Clouds

As described above, data layout of the vertices, complexity of the transfer function, and implementation of the geometry generation vary with the specific use case in which *attributed vertex clouds* are applied. They also

depend on the level of interactivity intended for the target application. For our examples, the transfer function and geometry generation implementations are distributed over vertex, tessellation, and geometry shader stages.

In the following, we present a number of use cases and explain the different implementations of AVCs in these contexts. Also, we provide a comparison of the proposed AVC approaches with an implementation of the same example using hardware instancing techniques. To illustrate the general concept, we present block worlds. As examples from the visualization domain, cuboids, arcs, polygons, and trajectories are described. All examples presented in this section can be accessed as code and executable with the demos provided[1]. They are based on OpenGL 4.0 and GLSL as shading language, some of them are compatible with OpenGL 3.2.



1.3.1 Block Worlds

The scene of our example block world contains equally-sized blocks of differing types that are arranged in a regular, three-dimensional grid. Conceptually, each grid cell may contain a block, thus a block can be identified through its position in the grid. Each block type is visually distinguishable through a different texture that is applied on its surface.

An implementation using hardware instancing [Carucci and Studios 05] would prepare a triangle strip with normalized positions and normal-vectors. Per-instance data passes the block position in the grid and the type to the rendering pipeline. During the vertex shader stage (Listing 1.1), each vertex is converted into actual world space coordinates using the normalized positions, the block position in the grid, and the size of each block (passed as uniform). The normalized positions and the triangle normal-vectors are also passed to the fragment shader to compute texture coordinates and perform texture lookup and shading.

```
1 uniform mat4 viewProjection;
2 uniform float blockSize;
3 in vec3  in_vertex; // instancing template
4 in vec3  in_normal; // instancing template
5 in ivec4 in_positionAndType; // per instance data
6 flat out vec3 g_normal;
7 flat out int  g_type;
```

[1] Hosted on https://github.com/hpicgs/attributedvertexclouds

```
8  out vec3 g_localCoord;
9
10 void main() {
11     gl_Position = viewProjection
12         * vec4((in_vertex + vec3(in_positionAndType.xyz)) * blockSize, 1.0);
13
14     g_normal = in_normal;
15     g_type = in_positionAndType.w;
16     g_localCoord = in_vertex * 2.0;
17 }
```

Listing 1.1. Block world vertex shader using hardware instancing. The per-instance data is packed into a single input.

To encode the same scene as an attributed vertex cloud, we use one vertex per block. The vertex contains the block position in the grid and its type. During the vertex shader execution (Listing 1.2), the center of the block is computed by the position in the grid and the size of each block (passed as uniform). In the geometry shader (Listing 1.3), the center and the size of each block is used to compute the corners and emit the triangle strip for the rasterization. This triangle strip has normal-vectors and normalized positions attached and is thus usable for the same fragment shader.

```
1  in ivec4 in_positionAndType; // per instance data
2  out int v_type;
3
4  void main() {
5      gl_Position = vec4(in_positionAndType.xyz, 1.0);
6
7      v_type = in_positionAndType.w;
8  }
```

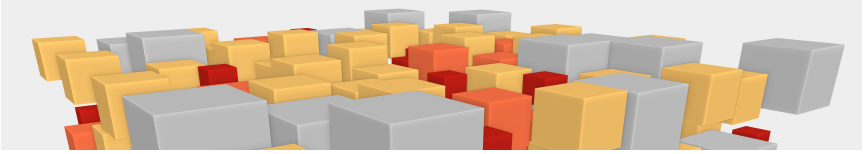
Listing 1.2. Block world vertex shader using AVCs. The per-instance data is packed into a single input.

```
1  layout (points) in;
2  layout (triangle_strip, max_vertices = 14) out;
3  uniform mat4 viewProjection;
4  uniform float blockSize;
5  in int v_type[];
6  flat out vec3 g_normal;
7  flat out int g_type;
8  out vec3 g_localCoord;
9
10 void emit(in vec3 position, in vec3 normal, in vec3 localCoord) {
11     gl_Position = viewProjection * vec4(position, 1.0);
12     g_normal = normal;
13     g_type = v_type[0];
14     g_localCoord = localCoord;
15     EmitVertex();
16 }
17
18 void main() {
19     vec3 center = gl_in[0].gl_Position.xyz * blockSize;
```

```
20      vec3 llf = center - vec3(blockSize) / vec3(2.0);
21      vec3 urb = center + vec3(blockSize) / vec3(2.0);
22
23      emit(vec3(llf.x, urb.y, llf.z), POSITIVE_Y, vec3(-1.0,  1.0, -1.0));
24      emit(vec3(llf.x, urb.y, urb.z), POSITIVE_Y, vec3(-1.0,  1.0,  1.0));
25      emit(vec3(urb.x, urb.y, llf.z), POSITIVE_Y, vec3( 1.0,  1.0, -1.0));
26      emit(vec3(urb.x, urb.y, urb.z), POSITIVE_Y, vec3( 1.0,  1.0,  1.0));
27      emit(vec3(urb.x, llf.y, urb.z), POSITIVE_X, vec3( 1.0, -1.0,  1.0));
28      emit(vec3(llf.x, urb.y, urb.z), POSITIVE_Z, vec3(-1.0,  1.0,  1.0));
29      emit(vec3(llf.x, llf.y, urb.z), POSITIVE_Z, vec3(-1.0, -1.0,  1.0));
30      emit(vec3(llf.x, urb.y, llf.z), NEGATIVE_X, vec3(-1.0,  1.0, -1.0));
31      emit(vec3(llf.x, llf.y, llf.z), NEGATIVE_X, vec3(-1.0, -1.0, -1.0));
32      emit(vec3(urb.x, urb.y, llf.z), NEGATIVE_Z, vec3( 1.0,  1.0, -1.0));
33      emit(vec3(urb.x, llf.y, llf.z), NEGATIVE_Z, vec3( 1.0, -1.0, -1.0));
34      emit(vec3(urb.x, llf.y, urb.z), POSITIVE_X, vec3( 1.0, -1.0,  1.0));
35      emit(vec3(llf.x, llf.y, llf.z), NEGATIVE_Y, vec3(-1.0, -1.0, -1.0));
36      emit(vec3(llf.x, llf.y, urb.z), NEGATIVE_Y, vec3(-1.0, -1.0,  1.0));
37      EndPrimitive();
38  }
```

Listing 1.3. Block world geometry shader using AVCs. The 14 emitted vertices build a full block triangle strip.

When comparing the instancing implementation to the AVC implementation, the main difference is encoding and application of the cube template. Using hardware instancing, the geometry is encoded using a vertex buffer that is input to the vertex shader stage. Using AVCs, the geometry is generated during the geometry shader stage. This results in fewer attributes passing the vertex shader stage.



1.3.2 Colored Cuboids

When using cuboids for rendering (e.g., for rectangular treemaps [Trapp et al. 13]), these cuboids may contain a position, an extent, and a color. The color is typically encoded as a scalar value that is converted to a color using a gradient.

Similar to the blocks of a block world, an instancing implementation would provide a cuboid triangle strip with normalized positions and normal-vectors (refer to the vertex shader in Listing 1.4). However, the actual position, extent, and color value are per-instance data. The color is computed using a texture lookup during the vertex shader stage. The inputs for the rasterization are the triangle strip with the attached color and normal-vector attributes.

The same scene encoded in an AVC would use the same per-instance data with adjusted vertex shader (Listing 1.5), as the target geometry

is generated during the geometry shader stage (Listing 1.6). Therefore, the provided position and extent attributes are used to compute the eight corners of the cuboid and a triangle strip is emitted containing the vertex positions and the attached normal-vectors and color attributes (Figure 1.2).

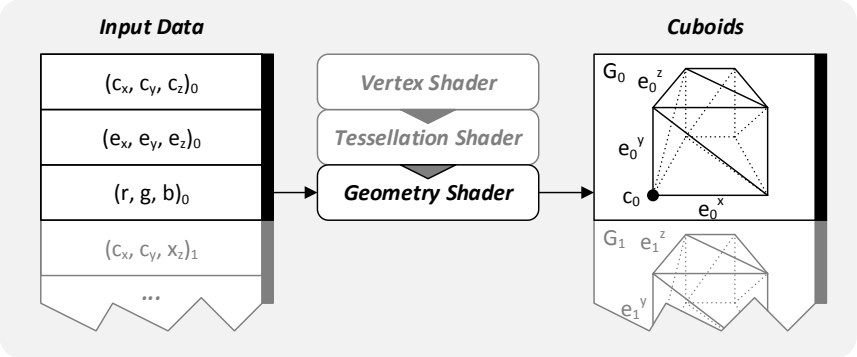


Figure 1.2. Concept of generating cuboids from an AVC. The center, extent, and color of each attributed vertex is used to emit one triangle strip with attached normal-vectors and colors.

```
1  uniform mat4 viewProjection;
2  in vec3  in_vertex; // instancing template
3  in vec3  in_normal; // instancing template
4  in vec3  in_position; // per instance data
5  in vec3  in_extent; // per instance data
6  in float in_colorValue; // per instance data
7  uniform sampler1D gradient;
8  flat out vec3 g_color;
9  flat out vec3 g_normal;
10
11 void main() {
12     gl_Position = viewProjection * vec4(in_vertex * in_extent + in_position, 1.0);
13     g_color = texture(gradient, in_colorValue).rgb;
14     g_normal = in_normal;
15 }
```

Listing 1.4. Cuboids vertex shader using hardware instancing.

```
1  in vec3  in_position; // per instance data
2  in vec3  in_extent; // per instance data
3  in float in_colorValue; // per instance data
4  uniform sampler1D gradient;
5  out vec3 v_extent;
6  out vec3 v_color;
7
8  void main() {
9      gl_Position = vec4(in_position, 1.0);
10     v_extent = in_extent;
```

```
11      v_color = texture(gradient, in_colorValue).rgb;
12      v_height = in_heightRange.y;
13  }
```

Listing 1.5. Cuboids vertex shader using AVCs.

```
1   layout (points) in;
2   layout (triangle_strip, max_vertices = 14) out;
3   uniform mat4 viewProjection;
4   in vec3 v_extent[];
5   in vec3 v_color[];
6   in float v_height[];
7   flat out vec3 g_color;
8   flat out vec3 g_normal;
9
10  // Emits a vertex with given position and normal
11  void emit(in vec3 position, in vec3 normal);
12
13  void main() {
14      vec3 center = gl_in[0].gl_Position.xyz;
15      vec3 halfExtent = vec3(v_extent[0].x, v_height[0], v_extent[0].y) / vec3(2.0);
16      vec3 llf = center - halfExtent;
17      vec3 urb = center + halfExtent;
18
19      emit(vec3(llf.x, urb.y, llf.z), POSITIVE_Y);
20      // ...
21      // analoguous to geometry shader of the block world
22      // ...
23      emit(vec3(llf.x, llf.y, urb.z), NEGATIVE_Y);
24      EndPrimitive();
25  }
```

Listing 1.6. Cuboids vertex shader using AVCs. The emitted triangle strip is created analoguous to the block world but without the local coordinates.

1.3.3 Colored Polygons

A polygon is a two-dimensional geometry type that has a number of vertices larger than two. In visualization, this geometry type is often extruded into the third dimension using the same polygon as bottom and top face, assigning a uniform height and adding side faces. To encode such a geometry in a single vertex is not feasible as the number of vertices of the polygon may differ between different polygons. However, we propose an approach that stores a polygon using two buffers. The first contains attributes that are common for the vertices of a single polygon (e.g., height, center, color value). The second contains each vertex of the polygon with

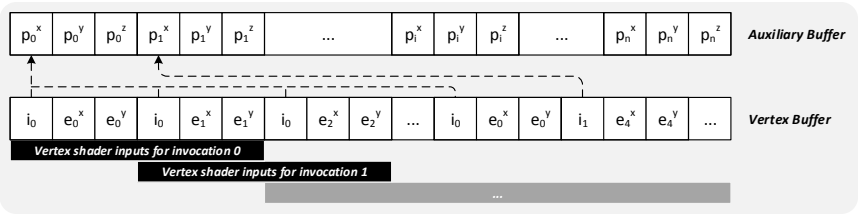


Figure 1.3. Polygon AVC buffer layout and vertex shader input management. One vertex shader invocation consumes two adjacent vertices as input but advances the by just one vertex for the next vertex shader invocation.

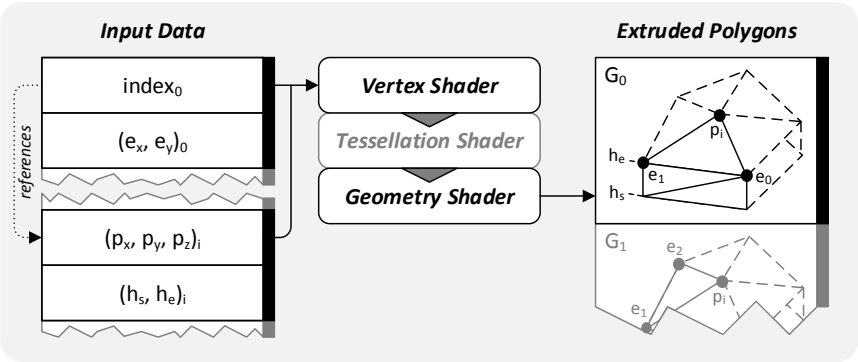


Figure 1.4. Concept of generating polygon-wedges from an AVC. One wedge is build up from the top and bottom triangle and the side face.

its two-dimensional position (the third dimension is stored as common attribute in the first buffer) and a reference to the polygon for the attribute fetching. The order of vertices in this buffer is relevant: adjacent vertices in the buffer has to be adjacent vertices in the polygon. To allow a closed looping over the vertices, we repeat the first vertex as the last one (Figure 1.3).

With this structure, the AVC can be rendered using the second buffer as vertex array buffer and the first buffer as auxiliary buffer that is accessed during the programmable rendering pipeline. We configure the vertex pulling stage to fetch two adjacent vertices from the buffer (handled as one vertex, refer to Listing 1.7), but advance only by one vertex for the next invocation (Figure 1.3). This way, we have access to the current and the next vertex and the common polygon data during the geometry generation which enables us to generate a polygon-*wedge* (Listing 1.8). To handle a pipeline start with two different referenced polygons for the two

vertices we stop the pipeline during the geometry shader stage. By looping over the complete list of vertices, all wedges of the polygon are generated and, as a result, the full polygon is rasterized.

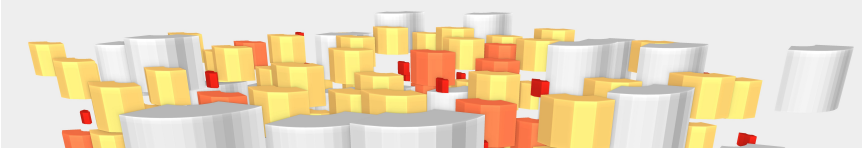
```
in vec2 in_start;
in int in_startIndex;
in vec2 in_end;
in int in_endIndex;
out vec2 v_start;
out vec2 v_end;
out int v_index;

void main() {
    v_start = in_start;
    v_end = in_end;
    v_index = in_startIndex == in_endIndex ? in_startIndex : -1;
}
```

Listing 1.7. Polygon vertex shader using AVCs. The index of the referenced polygon is checked to detect erroneuos wedge combinations and omit them in the geometry shader.

```
layout (points) in;
layout (triangle_strip, max_vertices = 6) out;

uniform mat4 viewProjection;
uniform samplerBuffer centerAndHeights;
uniform samplerBuffer colorValues;
uniform sampler1D gradient;
in vec2 v_start[];
in vec2 v_end[];
in int v_index[];
flat out vec3 g_color;
flat out vec3 g_normal;

// Emits a vertex with given position and normal
void emit(in vec3 pos, in vec3 n, in vec3 color);

void main() {
    // Discard erroneous polygons wedge combination
    if (v_index[0] < 0) return;

    vec4 centerAndHeight = texelFetch(centerAndHeights, v_index[0]).rgba;
    vec3 color = texture(gradient, texelFetch(colorValues, v_index[0]).r).rgb;
    vec3 cBottom = vec3(centerAndHeight.r, centerAndHeight.b, centerAndHeight.g);
    vec3 sBottom = vec3(v_start[0].x, centerAndHeight.b, v_start[0].y);
    vec3 eBottom = vec3(v_end[0].x, centerAndHeight.b, v_end[0].y);
    vec3 cTop = vec3(centerAndHeight.r, centerAndHeight.a, centerAndHeight.g);
    vec3 sTop = vec3(v_start[0].x, centerAndHeight.a, v_start[0].y);
    vec3 eTop = vec3(v_end[0].x, centerAndHeight.a, v_end[0].y);
    vec3 normal = cross(eBottom - sBottom, UP);

    emit(cBottom, NEGATIVE_Y, color);
    emit(sBottom, NEGATIVE_Y, color);
    emit(eBottom, NEGATIVE_Y, color);
    emit(sTop, normal, color);
    emit(eTop, normal, color);
    emit(cTop, POSITIVE_Y, color);
```

```
37      EndPrimitive();
38  }
```

Listing 1.8. Polygon geometry shader using AVCs to generate one polygon wedge.



1.3.4 Colored Arcs

An arc is typically used in visualization, e.g., for sunburst views of file systems. Such an arc can be represented using its center, inner and outer radii, the height range, and a color value. Storing this information in an AVC, we can generate a solid geometry that connects the side faces of the arc directly. However, a rendering should regard the conceptually round nature of an arc. As a GPU rasterizes planar primitives, the state-of-the-art solution is subdivision of the curved surface to produce planar surfaces that are arranged in a curve. This can be performed using the tessellation shader stage. The result is a set of arc segments with different angle-ranges, which the geometry shader can use to generate the target geometry (Figure 1.5).

Tessellation Control Shader. The output of the tessellation control shader is the degree to which the input geometry should be tessellated. Additionally, user-defined attributes (per-vertex and per-patch) can be specified and passed to the evaluation shader. We use this shader to tessellate two vertices (conceptually a line, refer to Listing 1.9) with the start and

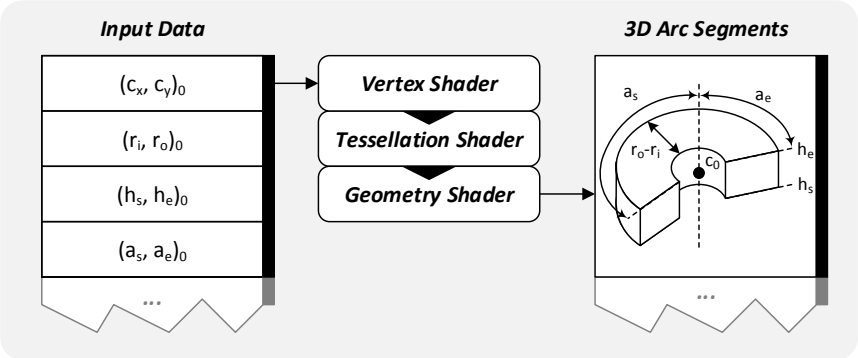


Figure 1.5. Concept of rendering arcs using an AVC. The tessellation shader creates arc lines that are extruded to arc segments using a geometry shader.

end angles as per-vertex attributes and the remaining ones as per-patch attributes. The use of both outer tessellation levels enables for higher amounts of subdivision of the arcs. With this configuration, the tessellator produces a set of subdivided lines.

```
layout (vertices = 2) out;

in Segment {
    vec2 angleRange;
    vec2 radiusRange;
    vec2 center;
    vec2 heightRange;
    vec3 color;
    int tessellationCount;
} segment[];

out float angle[];

patch out Attributes {
    vec2 radiusRange;
    vec2 center;
    vec2 heightRange;
    vec3 color;
} attributes;

void main() {
    angle[gl_InvocationID] = segment[0].angleRange[gl_InvocationID==0?0:1];

    if (gl_InvocationID == 0) {
        float sqrtTesslevel =
            clamp(ceil(sqrt(segment[0].tessellationCount)), 2.0, 64.0);
        gl_TessLevelOuter[0] = sqrtTesslevel;
        gl_TessLevelOuter[1] = sqrtTesslevel;

        attributes.radiusRange = segment[0].radiusRange;
        attributes.center = segment[0].center;
        attributes.heightRange = segment[0].heightRange;
        attributes.color = segment[0].color;
    }
}
```

Listing 1.9. Arcs tessellation control shader using AVCs. A minimum of two tessellation levels ensures that one segment cannot be both a left and right end.

Tessellation Evaluation Shader. This set of subdivided lines is reinterpreted as the start and end angles of an arc segment, interpolating the per-vertex angles of the prior shader. The per-patch attributes are assigned to each vertex of the resulting lines. Additionally, the first and last vertex of the full arc are flagged to allow the geometry shader to generate additional side-faces (Listing 1.10).

```
layout (isolines, equal_spacing) in;

in float angle[];
```

```
4
5  patch in Attributes {
6      vec2 radiusRange;
7      vec2 center;
8      vec2 heightRange;
9      vec3 color;
10 } attributes;
11
12 out Vertex {
13     float angle;
14     vec2 radiusRange;
15     vec2 center;
16     vec2 heightRange;
17     vec3 color;
18     bool hasSide;
19 } vertex;
20
21 void main() {
22     float pos = (gl_TessCoord.x + gl_TessCoord.y * gl_TessLevelOuter[0])
23         / float(gl_TessLevelOuter[0]);
24
25     vertex.angle = mix(angle[0], angle[1], pos);
26     vertex.radiusRange = attributes.radiusRange;
27     vertex.center = attributes.center;
28     vertex.heightRange = attributes.heightRange;
29     vertex.color = attributes.color;
30     float threshold = 1.0/(gl_TessLevelOuter[0]*gl_TessLevelOuter[1]);
31     vertex.hasSide = pos<threshold || pos>1.0-threshold;
32 }
```

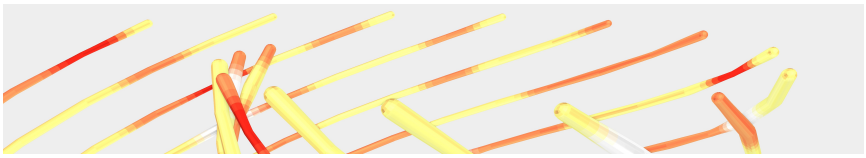
Listing 1.10. Arcs tessellation evaluation shader using AVCs. The position of the generated vertex is projected and used to subdivide the arc segment. It is also used to determine which vertices have side faces.

Geometry Shader. The geometry shader generates the arc segment target geometry. The start and end arc angles, inner and outer radii, as well as the lower and upper height values are used to compute the eight vertices of the arc segment (Listing 1.11). Depending on the `hasSide` property, the required faces are generated and emitted.

```
1  layout (lines) in;
2  layout (triangle_strip, max_vertices = 12) out;
3
4  in Vertex {
5      float angle;
6      vec2 radiusRange;
7      vec2 center;
8      vec2 heightRange;
9      vec3 color;
10     bool hasSide;
11 } v[];
12
13 uniform mat4 viewProjection;
14 flat out vec3 g_color;
15 flat out vec3 g_normal;
16
17 // Emits a vertex with given position and normal
```

```
18 void emit(in vec3 position, in vec3 normal);
19
20 vec3 circlePoint(in float angle, in float radius, in float height) {
21     return vec3(sin(angle), height, cos(angle))
22         * vec3(radius, 1.0, radius)
23         + vec3(v[0].center.x, 0.0, v[0].center.y);
24 }
25
26 void main() {
27     vec3 A = circlePoint(v[0].angle, v[0].radiusRange.x, v[0].heightRange.x);
28     vec3 B = circlePoint(v[1].angle, v[0].radiusRange.x, v[0].heightRange.x);
29     vec3 C = circlePoint(v[1].angle, v[0].radiusRange.y, v[0].heightRange.x);
30     vec3 D = circlePoint(v[0].angle, v[0].radiusRange.y, v[0].heightRange.x);
31     vec3 E = circlePoint(v[0].angle, v[0].radiusRange.x, v[0].heightRange.y);
32     vec3 F = circlePoint(v[1].angle, v[0].radiusRange.x, v[0].heightRange.y);
33     vec3 G = circlePoint(v[1].angle, v[0].radiusRange.y, v[0].heightRange.y);
34     vec3 H = circlePoint(v[0].angle, v[0].radiusRange.y, v[0].heightRange.y);
35
36     vec3 top = vec3(0.0, 1.0, 0.0);
37     vec3 bottom = vec3(0.0, -1.0, 0.0);
38     vec3 left = normalize(cross(E-A, D-A));
39     vec3 right = normalize(cross(F-B, C-B));
40     vec3 front = normalize(cross(B-A, E-A));
41     vec3 back = -front;
42
43     if (v[1].hasSide) {
44         emit(B, right);
45         emit(F, right);
46     }
47     emit(C, right);
48     emit(G, right);
49     emit(H, back);
50     emit(F, top);
51     emit(E, top);
52     emit(B, front);
53     emit(A, front);
54     emit(C, bottom);
55     emit(D, bottom);
56     emit(H, back);
57     if (v[0].hasSide) {
58         emit(A, left);
59         emit(E, left);
60     }
61
62     EndPrimitive();
63 }
```

Listing 1.11. Arcs geometry shader using AVCs. The code emits a triangle strip with six full rectangles forming a distorted cuboid. As it is impossible for both vertices of the input line to have the hasSide flag set, the maximum number of emitted vertices is 12 nethertheless.



1.3.5 Trajectories

For an information visualization example, we describe the application of AVCs to a visualization of movement trajectories [Buschmann et al. 15]. A trajectory consists of a number of connected tracking points, each of which contains the current position, a time stamp, and additional attributes such as the current speed or acceleration.

An interactive visualization of trajectories can be helpful for example for developing analysis and decision support systems. In such a system, one task would be to explore large data sets of trajectories, represented by plain lines for simplicity. Subsequently, trajectories can be selected to examine their movements in more detail. They are highlighted and displayed more prominently, for example as extruded tubes. As a final step of analysis, users might want to examine the attributes of individual tracking points. This can be achieved by rendering them using individual spheres, mapping color and radius to express attribute values.

The described use case can be implemented using AVCs as follows. The vertex buffer contains the nodes with their attributes and a reference to the the associated trajectory. In contrast to previous examples, a vertex does not directly describe the target geometry, but contains a mere data point which is transformed into a geometry later. In the tessellation and geometry shaders, the type of geometry is selected based on three factors: data attributes, distance to the camera, and interaction state. Unselected trajectories which are far away from the camera will be transformed into plain lines. Highlighted trajectories are tranformed into tubes, and for selected trajectories, nodes are transformed into individual spheres for attribute visualization. The current speed is mapped onto the color, while acceleration is mapped onto the radius.

To render the trajectories, a similar approach to the polygon rendering can be used. However, the vertex shader has three vertices as input, where the first and the third are used to check for the trajectory reference and the target geometry type. With these additional inputs, the tessellation and geometry shader for the currently processed trajectory node can generate geometry while taking the adjacent trajectory node geometries into account (Figure 1.6). The use of the tessellation shader stages allows for a curved appearance of the tube representation of trajectory nodes (refer to arc rendering with AVCs).

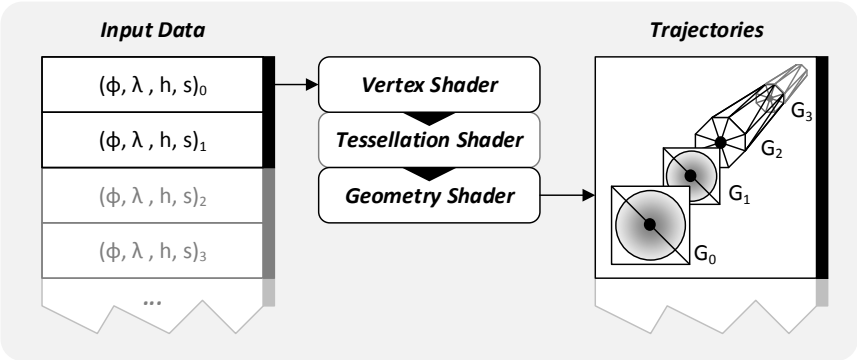


Figure 1.6. Concept of rendering trajectories using an AVC. Each data point gets tessellated and converted into splats, spheres, or tubes, depending on user input.

Due to the reduced memory footprints of AVCs, large numbers of trajectories can be rendered simultaneously. Also, the visualization is highly configurable without requiring any buffer updates. Therefore, it can be controlled by user interaction, e.g., by merely modifying uniform shader variables which control the geometry generation.

1.4 Evaluation

AVCs can improve the rendering of several types of geometry in terms of GPU memory usage, rendering and update performance. Therefore, we compare AVCs to alternative geometry representations and rendering pipelines for the same target geometry, namely full triangles lists (triangles), triangle strips, and hardware instancing using vertex attribute divisors (instancing). Specifically, evaluation is focused on the memory footprint of the geometry, the geometry processing performance, and the overall rendering performance.

Test Setup. Our test setup for the presented performance measurements is a Ubuntu 14.04 machine with a GeForce GTX 980 running at 1430 MHz maximum clock and 4GB VRAM. We used the official Nvidia driver version 355.06. During the tests, the application ran in full-screen mode with the native resolution of the monitor (1920 $\times$ 1200 pixels). The test scenes are block worlds with different numbers of blocks, ranging from 4096 (grid size of 16) to 10^6 (grid size of 100).

Technique	Static Data	Vertices	Blocksize	10^6 blocks
Triangles	$\perp$	36	1440 byte	1373 MiB
Triangle Strip	$\perp$	14	560 byte	534 MiB
Instancing	336 byte	1	16 byte	15 MiB
AVC	$\perp$	1	16 byte	15 MiB

Table 1.1. Memory consumption of different block world geometry representations. Each technique is compared regarding the required static data size, the number of vertices to encode one block, the resulting memory size of one block and a projection of the memory consumption of a block world with 10^6 blocks.

Technique	Vertex Shader Stage	Geometry Shader Stage	Rasterizer Stage
Triangles	10	$\perp$	11
Triangle Strip	10	$\perp$	11
Instancing	10	$\perp$	11
AVC	4	5	11

Table 1.2. Input component count of the used stages of the programmable rendering pipeline for the block world scene (e.g., a three-dimensional position has a component count of three).

Memory Footprint. To encode a block of a block world with a fixed size but differing type and position in a 3D grid, each of the four geometry representations have a differing space consumption on the graphics card (Table 1.1). The triangles representation requires 36 vertices per block with the position in the grid, a normal-vector, and the block type as attributes (40 bytes per vertex, 1440 bytes per block). The triangle strip representation requires the same per-vertex attributes but uses 14 vertices to encode the full block (40 bytes per vertex, 560 bytes per block). The instancing representation uses one triangle strip as instancing geometry (containing normalized vertex positions and normal-vectors) and passes the actual position in the grid and the block type as per-instance data (336 byte static data and 16 byte per-instance data, resulting in 16 byte per block). An AVC uses the same amount of per-instance data, but doesn't require the instancing geometry in memory as it is encoded in the geometry shader (16 byte per-vertex data, 16 byte per block). Although the instancing and the AVC representation share a small memory footprint of the geometry, the passed data through the programmable rendering pipeline differs (Table 1.2).

Rasterization Performance. To assess the overall performance of an application that renders the same scene (e.g., a block world) with different memory representations and rendering approaches, we use frames-per-second as a measure. We measured the time to render 1000 full frames

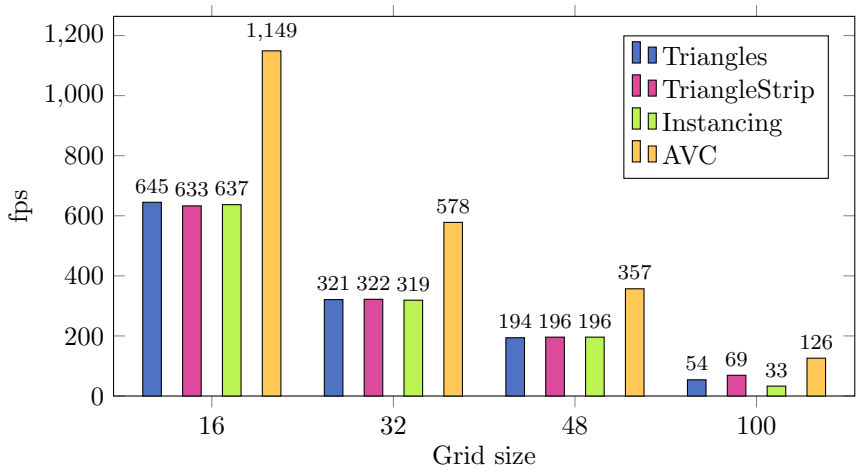


Figure 1.7. Frames per second comparison of block world implementations with different number of blocks.

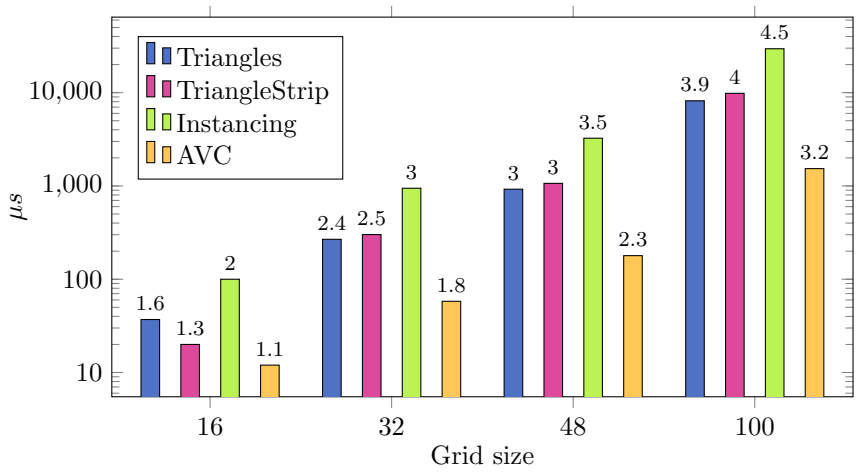


Figure 1.8. Geometry processing performance comparison of block world implementations with different number of blocks (logarithmic scale).

(same viewpoint, disabled vertical synchronization, disabled postprocessing, enabled rasterizer). Our measurements indicate that AVCs are faster to render in each scenario with up to doubled frames-per-second (Figure 1.7). The other techniques performs equally well for reasonable sized block worlds. We measure similar results for the cuboids demo scenes, but

there is a bigger variation within the other techniques. Since rendering a full frame requires additional work by both CPU and GPU that is equal between all techniques so we additionally measured the geometry processing in isolation.

Geometry Processing Performance. In order to measure the timings for the GPU to process the geometry and prepare the rasterization inputs, we disable the rasterizer and measure the time required for the draw call using query objects (Figure 1.8). We measured a significantly smaller processing time for the AVC technique and highly differing times for the other techniques where instancing performs worst. Although the times differ they, seem to have only small impact on the overall rendering performance (Figure 1.7). The results we measured for the cuboids scene were almost identical.

1.5 Conclusions

We propose attributed vertex clouds (AVC) as an efficient alternative to store geometry of simple gestalt for representation, rendering, and processing on the GPU. The concept utilizes attribute buffers that describe a target geometry. The attributes are processed and converted into the target geometry during the execution of the programmable rendering pipeline. The general concept has various potentials for rendering techniques and engines, since AVCs can be used in combination with explicit geometry representations. The main hardware requirement of this technique is a GPU with a programmable rendering pipeline that includes a geometry shader stage. A tessellation stage can be used to enhance the results. Performance measurements and comparisons using a block world test scene show a significant performance improvement on consumer hardware compared to explicit geometric representations and hardware instancing. With the compact memory representation of an AVC, attribute updates via GPGPU processing or buffer updates can be performed in the data domain instead of the target geometry domain.

The concept of attributed vertex clouds can be applied in general for all rendering techniques and domains in which image synthesis relies on large amounts of simple, parameterizable geometry. This includes games (e.g., block-worlds, vegetation rendering, sprite engines, particle systems), interactive visualization systems [Scheibel et al. 16] (e.g., information visualization, or scientific visualization), as well as generic rendering components (e.g., text rendering). Further improvement of this technique is a generalized rendering pipeline with reoccurring stages to enable further tessellation of the instantiated geometry.

Acknowledgements

This work was funded by the German Federal Ministry of Education and Research (BMBF) in the InnoProfile Transfer research group "4DnD-Vis" (http://www.4dndvis.de/) and BIMAP (http://www.bimap-project.de).

Bibliography

[Buschmann et al. 15] Stefan Buschmann, Matthias Trapp, and Jürgen Döllner. "Animated visualization of spatial–temporal trajectory data for air-traffic analysis." *The Visual Computer*, pp. 1–11.

[Carucci and Studios 05] Francesco Carucci and Lionhead Studios. "Inside geometry instancing." *GPU Gems 2*, pp. 47–67.

[Kemen 12] Brano Kemen. "Procedural Grass Rendering.", 2012. Available online (http://outerra.blogspot.de/2012/05/procedural-grass-rendering.html).

[Overvoorde 14] Alexander Overvoorde. "Geometry Shaders.", 2014. Available online (https://open.gl/geometry).

[Purnomo et al. 05] Budirijanto Purnomo, Jonathan Bilodeau, Jonathan D. Cohen, and Subodh Kumar. "Hardware-compatible vertex compression using quantization and simplification." *Proc. ACM Graphics Hardware*, pp. 53–61.

[Rákos 10] Daniel Rákos. "Instance Culling using Geometry Shaders.", 2010. Available online (http://rastergrid.com/blog/2010/02/instance-culling-using-geometry-shaders).

[Riccio and Lilley 13] Christophe Riccio and Sean Lilley. "Introducing the Programmable Vertex Pulling Rendering Pipeline." *GPU Pro 4: Advanced Rendering Techniques*, pp. 21–37.

[Scheibel et al. 16] Willy Scheibel, Matthias Trapp, and Jürgen Döllner. "Interactive Revision Exploration using Small Multiples of Software Maps." *Proc. of IVAPP*, pp. 131–138.

[Trapp et al. 13] Matthias Trapp, Sebastian Schmechel, and Jürgen Döllner. "Interactive rendering of complex 3d-treemaps." *Proc. of GRAPP*, pp. 165–175.