GPU-based Particle Systems

Introduction to Geometry Shaders and Transform Feedback

Programming – Computer Graphics

Contents

- Particle Systems
  - CPU-based
  - GPU-based
- Geometry Shaders
  - Input and Output
  - Writing Geometry Shaders
  - Points to Quads
- Transform Feedback

CPU-based Particle Systems

- So far we have covered particle systems that are CPU-based
  - The particles update on the CPU and send Quads to the GPU for rendering
- This can be slow due to the time spent updating hundreds to thousands of particles and updating vertex buffers for rendering

GPU-based Particle Systems

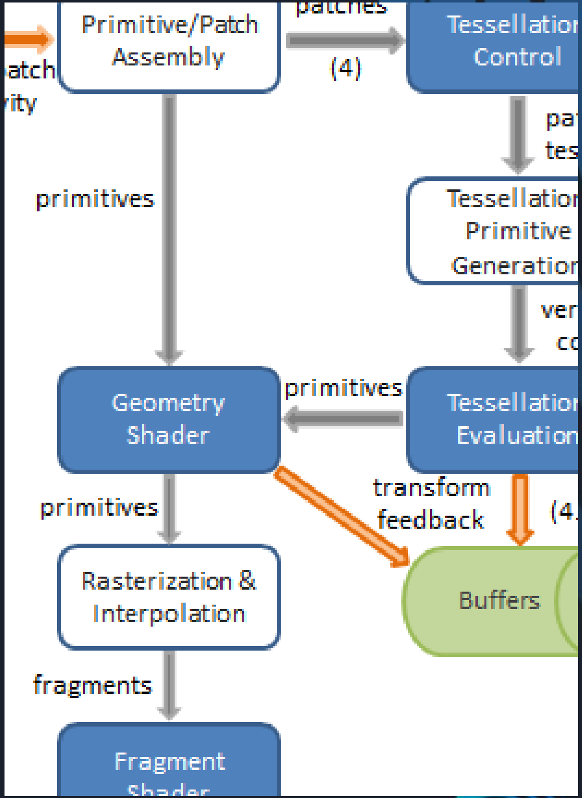
- The GPU has advanced quite a bit over the years
  - From simple Vertex and Fragment Shaders that rendered to the Back Buffer, to multiple shader stages and the ability to output to other targets
  - Multiple processing cores ranging into the thousands

- There are multiple ways we could take advantage of this processing power for our particle systems
  - Use the GPU to update our particles in a Vertex Shader and return the result using Transform Feedback
  - Use the GPU to render Points with a single vertex rather than 4-6 vertices for a Quad, making use of the Geometry Shader to turn our Points into Quads for us
  - Use the Geometry Shader to automatically Billboard the Quads for us

- Introducing the Geometry Shader…

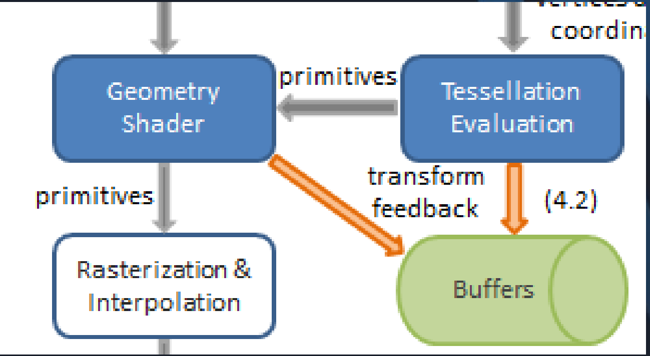
The Geometry Shader Stage

- An optional programmable stage
- Executed for each primitive in a mesh
- Receives the vertex data for all vertices used in the primitive
  - 1 for a point, 2 for a line, 3 for a triangle
- Can output more or less primitives than it receives
  - Can even change the type of the primitives!
  - And can output 0 primitives!
  - Does not automatically output the received primitive

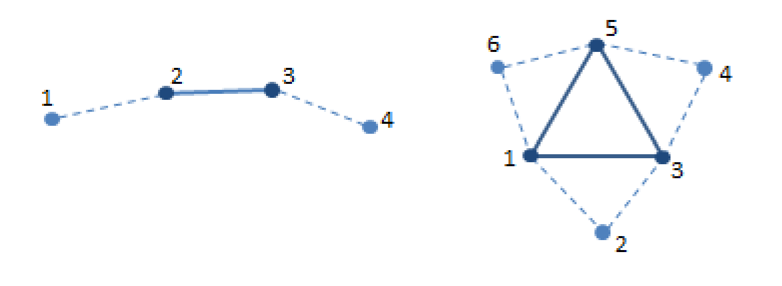


Geometry Shader Input and Output

- Despite there being multiple primitive types when rendering, the Geometry Shader only has 3
  - Points
  - Lines
  - Triangles
- It can also access adjacency information for primitives near the one being processed
  - Lines with adjacency and Triangles with adjacency
- It also has a limited set of output primitives
  - Points
  - Line Strips
  - Triangle Strips
- The Geometry Stage has one other advanced feature; it can output the processed vertices back to the application, without sending them to the Rasteriser to be drawn, through a method called Transform Feedback



Geometry Shader Input and Output



- Adjacency primitives are available when rendering primitives with adjacency information
  - These can be used instead of Line and Triangle primitives, but require more vertices per primitive
- The input adjacency primitives are as follows
  - Lines consist of 4 vertices rather than 2, with the 2nd and 3rd being the line itself, and the 1st and 2nd being the preceding line and 3rd and 4th being the following line
  - Triangles consist of 6 vertices with 1st 3rd and 5th being the triangle itself and the other points combine with the edges

Writing Geometry Shaders

- Writing Geometry Shaders is similar to the other shader stages, except that it requires a bit of extra work

```
layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;
```

  - You must define the input primitive layout and the output primitive layout
  - You must also define the maximum number of output vertices

Writing Geometry Shaders

- Here is an example of a simple Vertex Shader and Geometry Shader
  - The Geometry Shader is receiving triangle primitives and outputting triangle strip primitives with a max of 3 vertices, so just 1 triangle
- During the shader you must notify when a vertex has been fully defined
  - EmitVertex() notifies the shader that elements of the current vertex have been set

```
#version 410

in vec4 position;

uniform mat4 worldmatrix;
uniform mat4 projectionViewMatrix;

void main( ) {
	gl_Position = projectionViewMatrix * worldMatrix * position;
}
```

Vertex Shader

```
#version 410

layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

void main( ) {
	gl_Position = gl_in[ 0 ].gl_Position;
	EmitVertex();
	gl_Position = gl_in[ 1 ].gl_Position;
	EmitVertex();
	gl_Position = gl_in[ 2 ].gl_Position;
	EmitVertex();
}
```

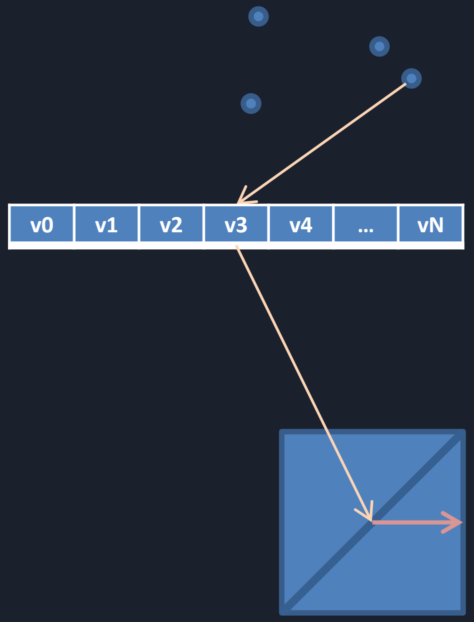
Geometry Shader

aie
SPECIALIST EDUCATORS IN
GAMES, ANIMATION & FILM VFX

Particle Points to Quads

- With particles, one simple application of the Geometry Shader is turning a single Vertex, rendered as a Point primitive, into a Quad primitive using a triangle strip
  - Update particles on the CPU as usual
  - Instead of creating billboarded quads for each particle point just send the array of particles to the GPU as a vertex buffer, rendered as Point primitives
    - 1 vertex per quad / sprite
  - In the Geometry Shader we turn the point into a billboarded triangle strip primitive

- This saves us CPU processing power and splits the task of billboarding the particles across all of the GPU's processing cores



Particle Points to Quads

- The following Geometry Shader demonstrates turning a single point into a billboarded triangle strip of 4 vertices
  - Rather than transform the position by the Projection and View matrices within the Vertex Shader we do that within the Geometry Shader
  - We still transform the vertex by the World / Model matrix within the Vertex Shader

```
#version 410

layout(points) in;
layout(triangle_strip, max_vertices = 4) out;

uniform float size;
uniform vec3 cameraPosition;
uniform mat4 projectionViewMatrix;

void main( ) {
	float halfSize = size * 0.5f;

	vec3 corners[4];
	corners[0] = gl_in[0].gl_Position.xyz + vec3( -halfSize, halfSize, 0 );
	corners[1] = gl_in[0].gl_Position.xyz + vec3( halfSize, halfSize, 0 );
	corners[2] = gl_in[0].gl_Position.xyz + vec3( -halfSize, -halfSize, 0 );
	corners[3] = gl_in[0].gl_Position.xyz + vec3( halfSize, -halfSize, 0 );

	mat3 billboard;
	billboard[2] = normalize( cameraPosition - gl_in[0].gl_Position.xyz );
	billboard[0] = cross( vec3(0,1,0), billboard[2] );
	billboard[1] = cross( billboard[2], billboard[0] );

	for ( int i = 0 ; i < 4 ; ++i ) {
		gl_Position = projectionViewMatrix * vec4( billboard * corners[ i ], 1 );
		EmitVertex();
	}
}
```

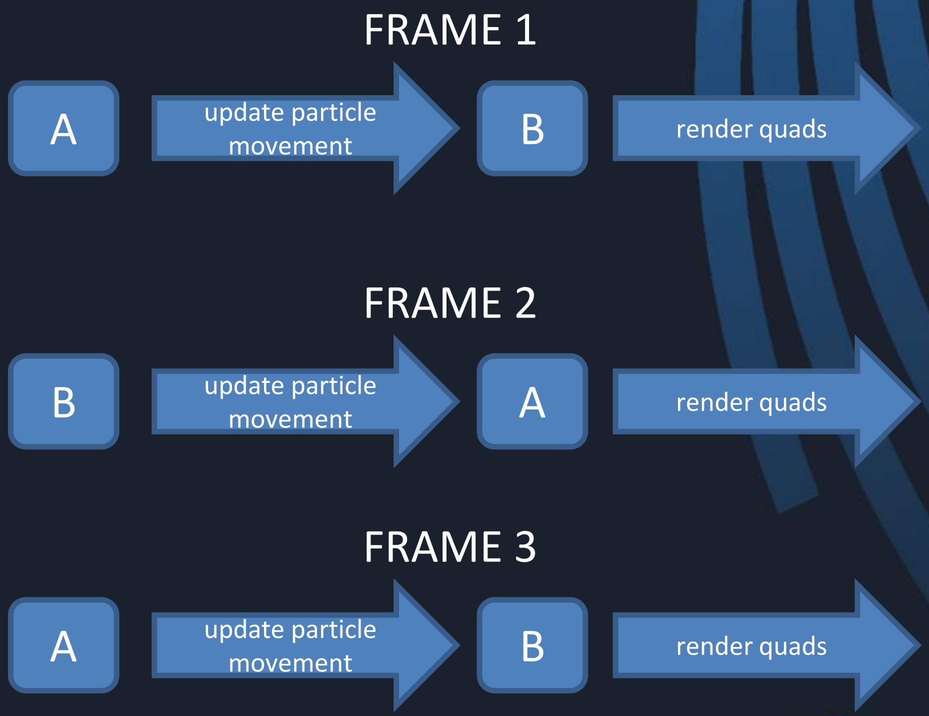
Transform Feedback

- The ability for us to output the vertices from the GPU before they are rasterised was also added at the same time as the Geometry Shader
  - Called Stream Output in DirectX and Transform Feedback in OpenGL
  - We can even disable the Rasterisation step and thus the Fragment Shader as well

- This means that we can process vertices through the Vertex Shader, Tessellation Shaders and Geometry Shader, then return the results to the CPU
  - The Tessellation Shaders and Geometry Shader are still optional
  - We bind a buffer to receive the output
  - We can then use this buffer like any other Vertex Buffer Object, rendering it like usual

Transform Feedback

- What this means for Particle Systems
  - In addition to moving the Billboarding step to the GPU we can move the updating of the particles to the GPU
  - This way the process is spread over all of the GPU's processing cores
- Using two buffers (because you can't have a buffer as an input AND output at the same time) we could
  - Render buffer A as points and update their movement in a shader
  - Output into buffer B the points that were processed from buffer A
  - Render buffer B as points and turn them into Quads for display
  - Next frame repeat the process, but start with buffer B outputting into buffer A instead



aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Updating on the GPU

- We can update the particles in any shader stage before the Transform Feedback takes place
  - Vertex Shader, Tessellation Shaders or the Geometry Shader
- Using the Geometry Shader we can increase or decrease particle counts
  - Dead particles can be ignored and not output for example
- As we generally need to know how many vertices will be in a Vertex Buffer, having the value increase or decrease at first sounds problematic
  - New API calls were added to be able to draw a returned buffer with the GPU keeping track of the vertex count itself

Summary

- Particle Systems can be almost entirely moved to the GPU
  - CPU time is saved for other features, such as Artificial Intelligence
  - Particles can be Billboarded within a Geometry Shader
  - Using Transform Feedback allows us to update particle positions on the GPU
- The Geometry Shader is a shader stage that receives all the information for a primitive, rather than just a single Vertex or Pixel
  - For example, all 3 vertices for a triangle would be received by the Geometry Shader when rendering triangles
- The Geometry Shader can do various last minute processing of primitives before they are sent to the Rasteriser and Fragment Shader
  - Or we can disable Rasterisation and just output processed vertices back to the CPU via Transform Feedback

Further Reading

- Wolff, D, 2013, *OpenGL 4 Shading Language Cookbook*, 2nd Edition, PACKT Publishing

- Haemel, N, Sellers, G & Wright, R, 2014, *OpenGL SuperBible*, 6th Edition, Addison Wesley

SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX